**MASTER EN SCIENCES INFORMATIQUES**

# Interaction software - firmware

# Choix de séquences de programme à implémenter en firmware

Jacquemart, Y.

*Award date:*
1975

*Awarding institution:*
Universite de Namur

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX-NAMUR
INSTITUT D'INFORMATIQUE

ANNEE ACADEMIQUE 1974 - 1975



# *INTERACTIØN SØFTWARE-FIRMWARE*

## CHØIX DE SEQUENCES DE PRØGRAMME A IMPLEMENTER EN FIRMWARE

**Y. JACQUEMART**

Mémoire présenté en vue de l'obtention
du grade de Licencié et Maître en Informatique.

**JURY : M. J. BRUNIN.**

C'est avec plaisir que nous remercions toutes les personnes qui ont contribué à la réalisation de ce travail.

Notre reconnaissance s'adresse particulièrement à Monsieur J. BRUNIN, directeur de ce mémoire, pour les réflexions et les critiques qu'il a bien voulu formuler.

Nous remercions également Monsieur J. DEMARTEAU pour l'intérêt évident qu'il a porté à ce travail; ses précieux conseils et ses encouragements nous ont permis d'approfondir toujours plus notre sujet.

Nous tenons aussi à exprimer notre gratitude à Messieurs BIENVENU et CASSONNET de la Compagnie Honeywell-Bull pour l'accueil chaleureux qu'ils nous ont réservé durant notre stage et les renseignements qu'ils ont pu nous procurer.

Nous signalons enfin le soin particulier qu'a apporté Madame WIBIN à la mise en pages de ce travail; nous lui en sommes profondément reconnaissants.

NAMUR, le 8 août 1975.

# TABLE DES MATIERES

---

# CHAPITRE 0

## INTRODUCTION.

0-0 Dans ce chapitre, nous nous proposons de définir l'orientation de notre travail.:

- en suivant rapidement l'évolution des ordinateurs, nous commencerons par rappeler ce qu'est la micro-programmation et dans quel but elle a été créée (0-1).

- nous décrirons ensuite deux nouvelles voies de recherche qu'elle a ouvertes (0-2, 0-3).

- nous situerons enfin notre application par rapport à ces deux voies de recherche (0-4).

0-1. Par définition, tout ordinateur est toujours constitué (V. fig. 0-1) :

- d'une partie électronique figée (hardware) et
- d'une partie programmable qui lui élargit son champ d'application.

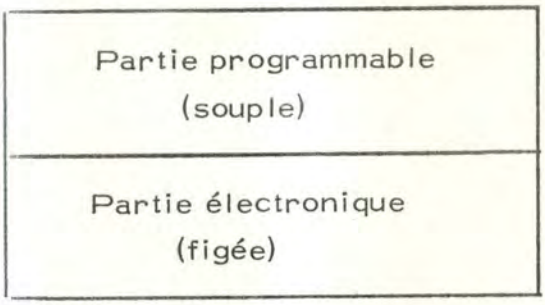



Fig. 0-1

Dans les premiers ordinateurs le langage machine (1) formait l'interface entre ces deux parties; il était défini et figé avant la construction du hardware et sur des critères purement utilitaires; le hardware était alors formé de séquenceurs déclenchés par les codes opérations des instructions machine; ces séquenceurs 'réveillaient' aux moments voulus les différentes unités du hardware; cette optique présentait deux inconvénients :

- nécessité de figer au départ et définitivement le jeu d'instructions machine (2);
- difficulté de conception d'un tel hardware (3).

Vers 1951, Wilkes systématisa l'organisation du hardware en introduisant le concept de micro-programmation (R 18).
Il défendait l'idée que l'on pouvait considérer l'exécution d'une instruction machine comme l'exécution d'un certain nombre de transferts synchronisés d'informations de registre - à - registre à travers des unités effectuant certaines opérations (4); chaque étape de transfert pouvait être comprise comme l'exécution d'une instruction (appelée micro-instruction) sur une machine inconnue du

---

(1) nous définirons le 'langage machine' comme le langage le plus élémentaire dans lequel l'utilisateur peut programmer ses traitements; cette définition reste assez vague; en effet, dans ce chapitre, nous verrons varier le niveau de ce langage avec l'évolution des idées qui se sont développées.

(2) le hardware est en effet non restructurable au niveau de l'utilisateur et difficilement restructurable au niveau du constructeur.

(3) complexité du hardware au point de vue conception et matériel utilisé, d'où coût élevé.

(4) certains transferts se déroulent en parallèle, d'autres en série.

programmeur (appelée host machine) (V. fig. 0-2); l'étape d'exécution d'une instruction dans la machine utilisateur (Virtual machine) était alors constituée par un micro-programme (1); l'ensemble des micro-programmes formait un interpréteur micro-programmé du langage machine; par opposition à la partie programmée au niveau le plus haut (2), on désigna par firmware l'ensemble des micro-programmes d'une machine.

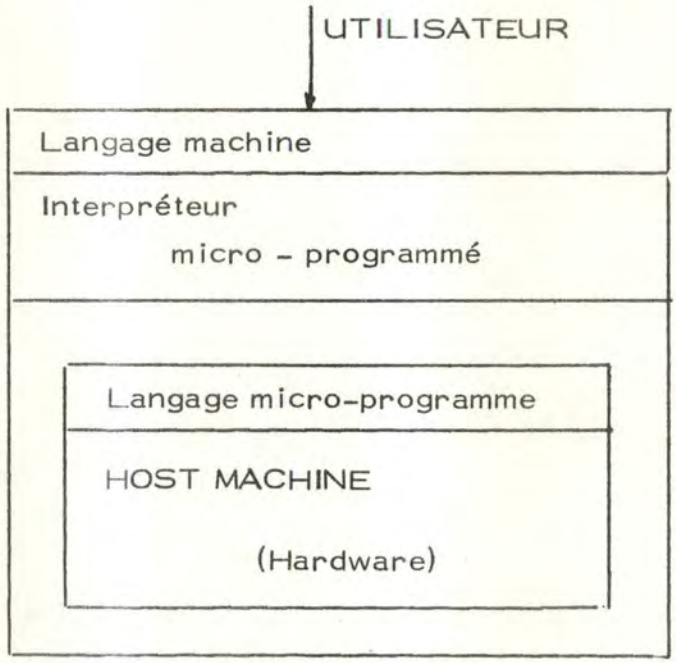



Fig. 0-2

Rem.

La micro-programmation offre de nombreux avantages; nous ne développerons pas ce point, mais le lecteur se référera avantageusement à REIGEL, FABER, FISHER (R12), ROSIN (R13) et BROADENT (R2).

0-2. Une des conséquences indirectes les plus importantes de l'emploi de la micro-programmation est sans doute :

la revalorisation des méthodes d'interprétation par rapport aux méthodes de compilation lorsqu'elles sont supportées par un hardware adéquat.

Sous peine de perdre sa souplesse et de demander des capacités gigantesques de mémoire, l'implémentation de la micro-programmation exige en effet l'emploi d'un interpréteur; or lorsque Wilkes émit ses idées sur la micro-programmation, la plupart des constructeurs rejetaient l'interprétation, basant leur argumentation sur leurs

---

(1) le micro-programme assure donc le séquencement des transferts.

(2) Software.

expériences software antérieures ou sur certaines caractéristiques des deux méthodes de traduction (V. fig. 0-3 et ses commentaires).

En vue de compenser la lenteur relative des interpréteurs tout en profitant des caractéristiques 'placé' qu'ils apportaient, on rechercha alors un moyen de les accélérer; c'est seulement plus tard (1), grâce à l'apparition sur le marché de nouvelles technologies de mémoires que l'on trouva la solution; les tailles relativement faibles des interpréteurs leur permettaient de jouir des temps d'accès très courts de certaines mémoires, alors qu'un programme utilisateur ne le pouvait en principe pas (2).

Pour mieux comprendre l'effet de l'introduction des mémoires rapides, nous regarderons la fig. 0-4; elle décrit l'exécution d'une instruction machine comme l'appel et l'exécution d'un micro-programme se trouvant en mémoire rapide; une grande partie de l'exécution du programme utilisateur se passe donc en mémoire rapide; grâce à l'interprétation, l'utilisateur profite donc, à la fois à travers son langage machine :

- d'un gain de place sur le programme objet grâce à l'interprétation.
- des temps d'accès très faibles de mémoires rapides.

Le bilan en temps n'est cependant pas nécessairement positif; il variera selon les considérations suivantes :

- plus la différence de vitesse entre les deux mémoires est grande, meilleur sera le bilan en temps;
- plus les micro-programmes situés en mémoire rapide seront longs (3), meilleur sera le bilan en temps (4);

---

(1) c'est ce qui explique l'apparition tardive sur le marché des premiers ordinateurs micro-programmés :
1958 : première description d'un ordinateur micro-programmé sur le modèle de Wilkes (R5).
1964 : première implantation sur IBM360 (R1, R15).

(2) la taille des mémoires est limitée par trois facteurs principaux (R16)
- facteur temps d'accès : pour une même technologie, plus une mémoire est grosse, plus les temps d'adressages et de lecture/écriture sont longs. Il y a donc un compromis entre la taille d'une mémoire et la vitesse d'accès aux informations.
- facteurs physiques : dissipation et proximité ...
- coût : les mémoires les plus rapides étant généralement les plus coûteuses (coût par bit).

(3) une limite sur la longueur est cependant imposée par la grandeur de la mémoire rapide; sinon, l'idéal serait d'avoir un programme utilisateur entièrement micro-programmé.

(4) ici, se pose tout le problème de structuration; en gros, l'idéal est d'avoir rempli la mémoire rapide des traitements les plus employés; cela pose le problème du choix des instructions machine; nous en parlerons en 0-3.

- plus le temps passé à la 'traduction dynamique' (1) est court, meilleur sera le bilan en temps.

Ces considérations posent donc le problème de la recherche du meilleur langage machine (Voir paragraphe 0-3).

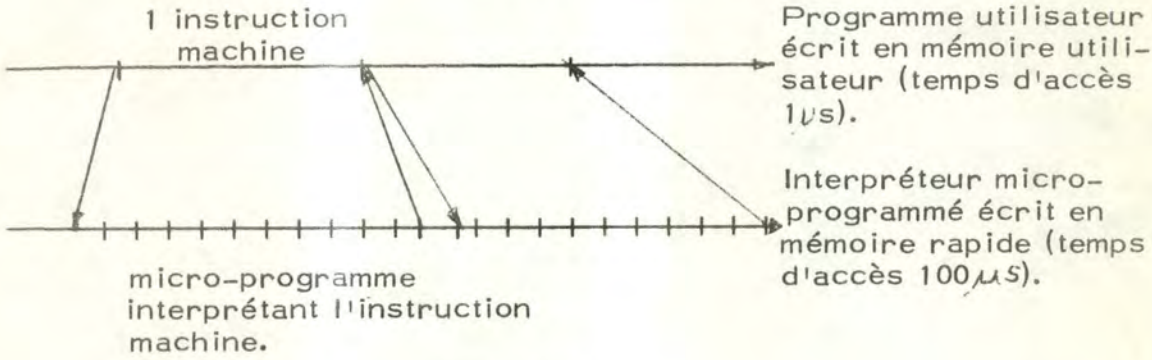



Fig. 0-4

Depuis peu, de nouvelles voies de recherche ont généralisé cette configuration initiale en une configuration possédant plusieurs niveaux d'interprétation, avec en parallèle des hiérarchies de mémoires, de programmes et de données (R14, R19).
Les réalisations dans ce domaine sont encore peu nombreuses; certains hardware ont fait les premiers pas vers de telles configurations; nous citerons :

- l'ordinateur QM1 de Nanodata Corporation (R22) ;
- et le projet EPRON des facultés N. D. de la Paix à Namur.

Remarques à propos de la terminologie employée :

- la mémoire rapide est généralement appelée mémoire de contrôle (CS = control store); elle peut être de deux types :
  - soit accessible uniquement en lecture (ROM = read only memory);
  - soit accessible également en écriture (WCS = writable control store).

---

(1) la traduction dynamique implique, en fait, l'exécution d'opérations qui ne sont pas nécessaires dans tous les cas, mais qui sont malgré tout conservées pour donner au micro-programme un interface standard (ex. : positionnement du code condition, addition d'un déplacement nul dans un calcul d'adresse ....).

Caractéristiques logiques

| | COMPILATION | INTERPRETATION |
|---|---|---|
| A | Traduction préliminaire à l'exécution | Traduction pendant l'exécution |
| B | Traduction globale | Traduction locale |
| C | Traduction statique | Traduction dynamique |
| | | |
| D | | Perte de temps à l'exploitation par rapport à une méthode de compilation (V. note 1 p. 0-5). |
| E | Le programme objet prend plus de place que par la méthode d'interprétation. | |

Fig. 0-3

Commentaires de la fig. 0-3.

A - la compilation se passe dans une phase préliminaire à l'exécution; elle n'est donc exécutée qu'une seule fois (1); l'interprétation par contre, est réalisée à chaque exécution.

B - la compilation est une traduction globale, c'est-à-dire qu'en principe chaque instruction est traduite en tenant compte du contexte dans lequel elle se trouve (2), l'interprétation associe à chaque instruction, au moment de l'exécution, un module prédéfini; elle est donc locale.

C - la compilation traduit une seule fois chaque instruction; elle est donc statique par rapport à l'interprétation qui traduit chaque instruction du programme autant de fois que l'on passe dessus à l'exécution (3).

D-E - résument les caractéristiques logiques finales de ces deux méthodes sans tenir compte de leur implémentation.

---

(1) lorsque le programme est prêt à être exploité.

(2) ce qui introduit la notion d'optimisation du code généré par un compilateur; en général, ces optimisations sont réduites, ce qui limite l'intérêt relatif de la compilation au point de vue temps d'exécution.

(3) c'est-à-dire en théorie de 0 à l'infini.

0-3. Une deuxième voie de recherche s'est ouverte à partir d'un avantage indiscutable de la micro-programmation : la souplesse d'adaptation du langage machine.
Celle-ci allait pousser les chercheurs à se poser un nouveau problème: comment adapter au mieux un langage machine pour qu'il satisfasse à la fois les deux considérations suivantes :

- qu'il soit 'naturel' pour le(s) compilateur(s) (1);
- qu'il soit efficace pour l'exécution des programmes exploitables (2).

Ce problème très complexe a déjà suscité de nombreuses recherches mais est encore loin d'être résolu.

La plupart des constructeurs ont adopté un langage machine unique, compromis entre les besoins des différents compilateurs; l'interpréteur figé et statique se trouvait alors dans une mémoire ROM; cette solution est cependant réalisée au détriment de l'efficacité du système; certains chercheurs se sont penchés sur la reconfiguration d'un langage machine en fonction de besoins particuliers; ces recherches ont montré que la rajoute de quelques nouvelles instructions plus complexes (3) à un langage machine existant peut améliorer considérablement le temps d'exécution des programmes; nous citerons à ce sujet une expérience tentée sur un IBM 360/40 (R6) et l'article de Tucker et Flynn (R17).

Encouragés par les résultats obtenus, un grand nombre de chercheurs s'orientèrent vers la construction d'un langage machine particulier pour chaque langage de haut-niveau; les interpréteurs restèrent donc figés, mais la configuration de la WCS devint dynamique (4); deux directions furent prises dans ce domaine :

---

(1) selon le choix du constructeur, un langage machine peut servir à tous les compilateurs, ou chaque compilateur peut avoir son langage machine.

(2) l'efficacité se résume en deux points :

- prendre le moins de place possible pour le programme exploitable;
- prendre le moins de temps possible pour l'exécution de ce programme.

(3) TUCKER et FLYNN (R17) décrivent :

- calcul d'adresses des éléments des Tableaux
- génération des suites de FIBONACCI,
- Algorithme de recherche en tables ,

d'autres exemples peuvent être cités :

- fonction de Tris sur GE58 (Honeywell Bull)
- Algorithme de compactage/décompactage de fichiers sur P7 (Honeywell Bull)
- Dispatcher micro-programmé sur P7
- Multiplication des matrices
- fonctions trigonométriques.

(4) cela pose de nouveaux problèmes de gestion de la WCS et de protection; ces problèmes sont esquissés par ROSIN (R13).

- Burroughts créa dans le B 1700 (R21) trois langages machine conventionnels en rapport direct avec les compilateurs COBOL, FORTRAN et RPG.
- Dans une autre direction, on prit pour langage machine les langages intermédiaires générés par la compilation (R3, R9) et en général la notation polonaise postfixée; nous citerons dans ce domaine (R8, R11, R20); remarquons tout de même que cette deuxième orientation réduit la compilation à un strict minimum.

L'efficacité d'un langage machine adaptable encouragea certaines firmes (1) à répondre aux besoins croissants de dynamicité au niveau de l'utilisateur (2); elles fournirent à celui-ci :

- un jeu d'instructions, standard et figé en ROM;
- une WCS et des moyens d'accès à cette WCS.

Malheureusement, malgré les recherches concernant des langages de micro-programmation de haut-niveau (R 7, R10), il reste difficile (3) pour un utilisateur non averti de micro-programmer efficacement; de telles configurations restent, en pratique, inutilisables.

Nous résumerons dans la Fig. 0-5 les différents types de configurations décrites; nous ferons remarquer que le passage de la mono-programmation à la multi-programmation dans des configurations où l'interpréteur est dynamique, pose deux types de problèmes :

- problèmes de sécurité ;
- problèmes de gestion de la WCS.

Dans notre application, nous nous limiterons à la mono-programmation ; nous laisserons au lecteur le soin de rechercher et de résoudre les problèmes nouveaux qui se poseraient dans une configuration de multi-programmation.

---

(1) Hewlett Packard (HP21MX, HP2100), Interdata (M85) et Varian 72 ...

(2) Cette dynamicité n'est prévue que sur des petits systèmes travaillant en mono-programmation; elle pose en effet un problème délicat de protection; celui qui à la possibilité de micro-programmer devient en effet le maître absolu de la machine.

(3) cette difficulté provient du fait que micro-programmer demande à l'utilisateur de se détacher de son problème logique pour se pencher sur des considérations d'implémentation dépendant uniquement de particularités du hardware.

| Caractéristiques de l'interpréteur chez l'utilisateur | Mono-programmation | Multi-programmation | Exemples |
|---|---|---|---|
| - statique (unique en ROM)<br>- figé (par le constructeur) | x | x | la plupart des systèmes |
| - Dynamique (plusieurs se partagent la WCS, d'où problème de gestion de la WCS)<br>- Figés (par le constructeur) | x | x | Burroughts |
| - 1ère partie statique figée<br>- 2ème partie dynamique modifiable (par l'utilisateur, d'où problèmes de protection) | x | | Varian... |

Fig. 0-5

0-4. Poursuivant la ligne de recherche tracée en 0-3, et tenant compte des considérations suivantes :

A - la plus grande dynamicité du code machine (et donc de l'interpréteur associé) semble apporter les meilleurs résultats au point de vue temps d'exécution (1) et place de stockage des programmes objets : la meilleure chose à faire, semble donc d'adopter le langage machine au niveau d'une application (1 programme);

B - obliger le programmeur normal à micro-programmer, c'est réduire les facilités qui lui sont offertes par les langages de haut-niveau en lui demandant de se détacher de son application logique pour se pencher sur des considérations souvent complexes et en rapport direct avec le hardware; de plus, pour les problèmes de protection, l'utilisateur ne peut avoir le droit de micro-programmer; dans une situation idéale, l'utilisateur sera donc totalement inconscient de ce qui se passe derrière le langage machine qu'il utilise;

---

(1) pourvu que les temps de gestion de la WCS ne deviennent pas trop importants; cette gestion peut être définie au niveau du programme ou de parties indépendantes de celui-ci (tâches, parties indépendantes d'une tâche) ; les temps de gestion dépendent à la fois de caractéristiques hardware et de la configuration des programmes; ne possédant aucun outil capable d'estimer de manière précise ces temps, nous avons été amenés à faire le choix le plus vraisemblable : une dynamicité de la WCS au niveau du programme.

nous avons été amenés à définir une nouvelle configuration contenant entre autre un interface software entre les traitements écrits dans un langage de haut-niveau et les micro-programmes à générer; nous décrirons ci-après cette configuration.

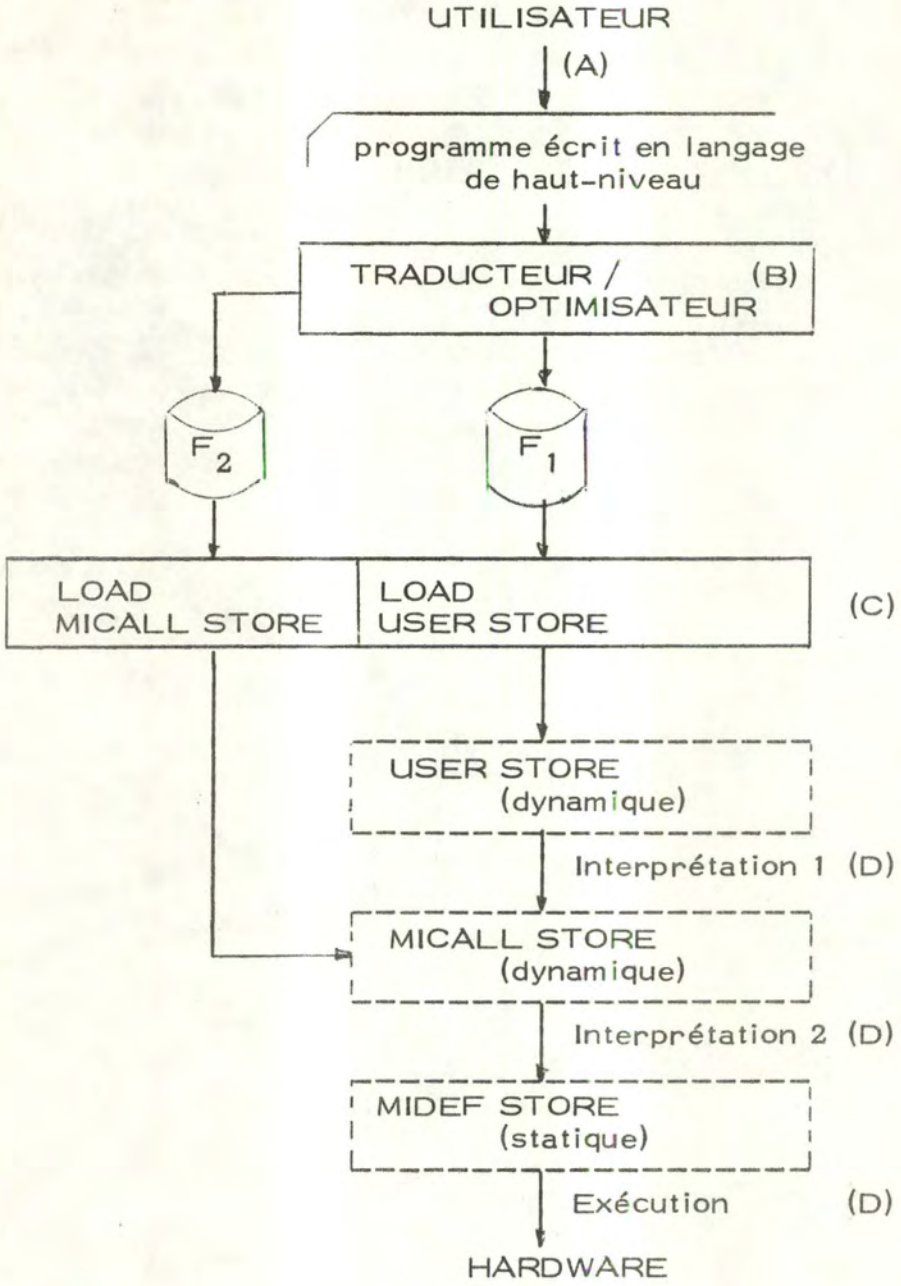



Fig. 0-6 (1)

(1) F1 contient le programme objet sous forme relogeable et écrit dans un langage machine dont l'interpréteur se trouve dans F2.

Dans une première étape (V. fig. 0-6) (A), l'utilisateur programme son traitement dans un langage de haut-niveau; à partir de ce programme un TRADUCTEUR/OPTIMISATEUR dont nous décrirons la logique de fonctionnement au chapitre 1, construit :

- un programme objet écrit dans un langage machine non défini à priori (fichier F1);
- un interpréteur de ce langage machine qui a pour but de le définir (fichier F2).

Lors de chaque exécution, le contenu des fichiers F1 et F2 est chargé (C) respectivement dans la USER STORE et la MICALL STORE. Le mécanisme d'exécution (D), représenté à la fig. 0-6 et décrit par J. Demarteau (R4), se compose de deux étapes d'interprétation auxquelles sont associés trois niveaux de mémoire, appelés (par ordre de vitesses croissantes) : USER STORE, MICALL STORE et la MIDEF STORE; le contenu exact de ces mémoires sera précisé au chapitre 1; sachons cependant que la MIDEF STORE a un contenu statique, et que les deux autres ont un contenu dynamique.

Le mécanisme s'appuie sur le courant de recherche décrit en 0-2 (1); les avantages que nous comptons en retirer sont les suivants :

- gain évident de place mémoire pour tous les programmes objets et un gain probable en temps d'exécution (V. page 0-4);
- facilité de génération de l'interpréteur dynamique par le traducteur/optimisateur en dégageant celui-ci de toutes les contraintes hardware (2).

Nous n'approfondirons pas plus la logique de ce mécanisme d'exécution, notre but étant de décrire le traducteur/optimisateur. Nous décrirons cependant brièvement l'implantation physique de ce mécanisme; elle résulte d'un compromis entre les nécessités de notre application et les possibilités d'un hardware disponible, celui du Varian 72 (R23, R24); celui-ci ne possède en effet que deux niveaux de mémoire : USER MEMORY et WCS; nous avons donc été obligés de partager la mémoire rapide WCS en deux parties logiques représentant la MICALL STORE et la MIDEF STORE, ce qui provoquera une certaine perte d'efficacité au point de

---

(1) concernant les hiérarchies de mémoires et de données associées au mécanisme d'interprétation.

(2) tous les problèmes concernant les simultanéités et les contraintes hardware peuvent être résolus manuellement et définitivement par un micro-programmeur, plus apte à résoudre efficacement ces problèmes qu'un programme spécialisé.
L'interpréteur ainsi figé sera mis dans la mémoire MIDEF.
La mémoire MICALL contiendra des appels aux micro-programmes prédéfinis se trouvant dans la mémoire MIDEF.

vue temps; on peut cependant espérer que la dégradation (1) ne sera pas trop importante; de toute façon l'avantage place subsiste et peut être un facteur très important sur un ordinateur ne possédant que peu de mémoire utilisateur.

---

(1) attention ! il s'agit d'une dégradation en temps par rapport aux performances attendues; mais il se peut que le bilan 'temps' global reste tout de même positif.

# CHAPITRE 1

## LES GRANDES ETAPES LOGIQUES DE L'APPLICATION ET SON INTEGRATION DANS UNE CONFIGURATION EXISTANTE

1-0. Dans la fig. 0-6, nous avions illustré l'idée générale que nous poursuivons, c'est-à-dire créer un software spécialisé (TRADUCTEUR/OPTIMISATEUR) capable de :

- traduire un programme écrit dans un langage de haut-niveau en un programme écrit dans un langage machine, à priori, non défini;
- de définir le langage machine et de générer l'interpréteur associé.

Ce software peut donc être divisé en deux parties logiquement distinctes mais interagissant entre-elles : la COMPILATION et l'OPTIMISATION; nous ne nous intéresserons pas à la partie COMPILATION; nous réutiliserons les compilateurs existant sur l'ordinateur VARIAN 72; cela imposera, bien sûr, certaines contraintes qui n'affecteront en rien les principes fondamentaux de notre application, mais nous obligeront à prendre certaines options.

Les buts de ce chapitre seront :

- de définir la structure de l'interpréteur à générer et d'exprimer les avantages que nous en attendons (1-1);
- de définir la logique générale de l'OPTIMISATEUR et de préciser les gains que nous espérons (1-2);
- de définir les relations entre l'OPTIMISATEUR et les COMPILATEURS existant et de justifier les options prises (1-3);
- de signaler quelques problèmes de fiabilité (1-4).

## 1-1. DEFINITION DE LA STRUCTURE DE L'INTERPRETEUR.

Le principe de la double interprétation que nous avons adopté trouve sa justification dans la structure même des interpréteurs micro-programmés couramment utilisés.
En effet, au niveau de son exécution logique, une instruction machine peut être décomposée en un nombre plus ou moins grand (1) d'étapes indépendantes (2), certaines étant identiques pour toutes les instructions machine d'un même type (3).
Lors de l'implémentation des instructions en micro-programmes, la structure logique se conserve plus ou moins (4), ce qui entraîne une certaine modularité de l'interpréteur micro-programmé.
L'exécution apparaît donc (Fig. 1-1) comme l'exécution d'un certain nombre de modules chaînés à l'intérieur de la mémoire de contrôle; ce chaînage possède deux caractéristiques :

- il est souple, en ce sens qu'il est conditionné par le code opération de l'instruction en cours (5);
- il est limité, en ce sens qu'un certain nombre de chaînages ont été prévus et que les autres ne peuvent être réalisés.

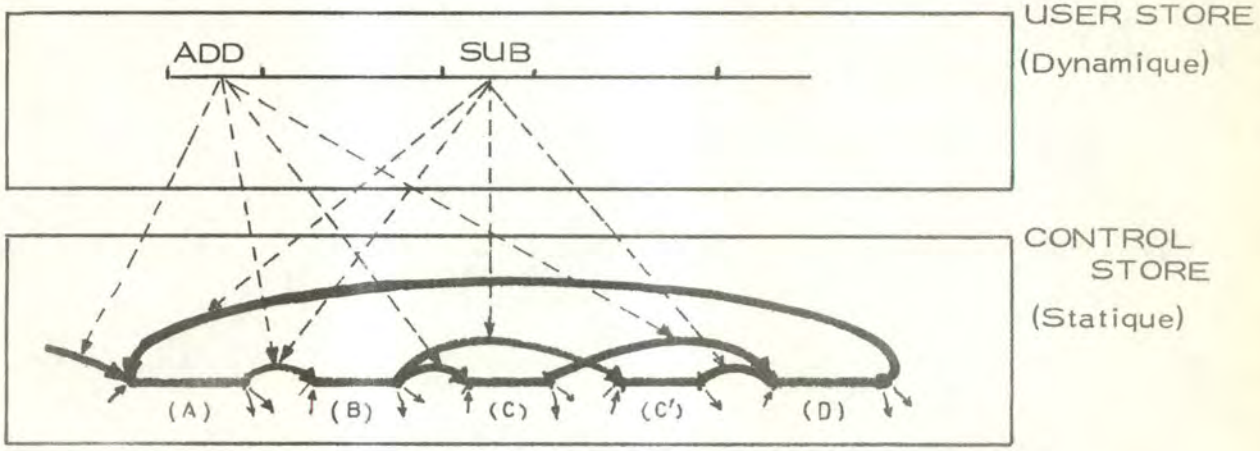



Fig. 1-1 : Interpréteur micro-programmé courant (V. légende)

---

(1) ce nombre dépend de la complexité des instructions machine.

(2) par exemple : calcul d'adresses, lecture des données, opérations particulières (Addition...), lecture de l'instruction suivante.

(3) par exemple : un langage machine ne possède qu'un nombre limité de formats d'adresses différents; les calculs de ces adresses seront donc standardisés.

(4) cela est surtout vrai dans les mini-ordinateurs ne possédant que peu de facilités hardware.

(5) par exemple, dans la fig. 1-1, selon que l'on aura une addition ou une soustraction, toutes les étapes seront communes sauf l'étape C/C'.

Légende : ⇢ conditionnement des branchements par le code opération de l'instruction.

→ chaînages proprement dits

⇉ le nombre de branchements possibles est limité

(A) calcul d'adresses (B) lecture de la donnée

(C) opération d'addition (C') opération de soustraction

(D) lecture de l'instruction suivante.

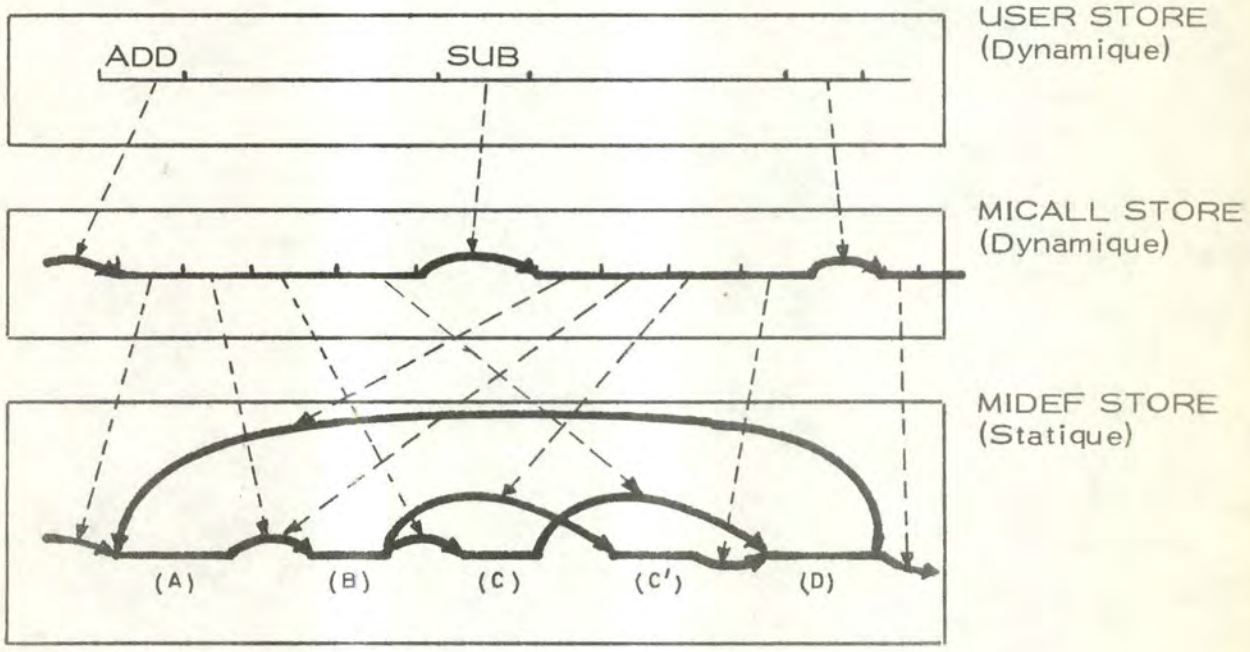



Fig. 1-2 : Interpréteur micro-programmé à deux étapes d'interprétation. (V. légende)

L'idée d'assouplir ce chaînage (1) ouvre la voie au principe de la double interprétation. Un ensemble de modules prédéfinis (2) et n'ayant, à priori, aucune relation physique entre-eux sont placés dans la mémoire MIDEF (V. fig. 1-2); lors d'une exécution, un programme spécialisé se trouvant dans la mémoire MICALL assure le conditionnement des chaînages.
Ce programme est en réalité une suite d'adresses des modules à chaîner; la souplesse désirée est donc acquise.

---

(1) cet assouplissement se fera en passant d'un chaînage limité une fois pour toute lors de la conception, à un chaînage illimité lors de la conception, mais limité au niveau de chaque programme par le contenu de la mémoire MICALL.

(2) ces modules seront au départ les modules constituant les instructions machine actuelles; dans la suite, on adaptera empiriquement ces modules aux besoins de nos applications; le software que nous allons décrire aidera à cette adaptation.

Ce mécanisme possède trois avantages fondamentaux :

1. Il permet d'avoir des instructions machine plus complexes et mieux en rapport avec notre application, ce qui entraîne un gain de place et de temps;
2. Il évite de gaspiller trop de place en mémoire rapide (1) malgré la complexité des traitements implémentables; il permet donc de profiter au maximum de la rapidité de cette mémoire;
3. Il permet une génération plus aisée et donc plus rapide par rapport à une génération directe de micro-programmes.

Pour profiter au mieux des avantages qu'offre l'ordinateur VARIAN, notre mécanisme sera un compromis entre les deux mécanismes précédents, c'est-à-dire :

- que nous conserverons un langage machine de base (2);
- que nous assouplirons ce deuxième langage machine grâce au deuxième mécanisme se trouvant dans la WCS.

---

(1) un traitement complexe entièrement micro-programmé prendrait beaucoup plus de place en mémoire rapide que le programme de conditionnement généré en mémoire MICALL.

(2) celui de l'ordinateur VARIAN dont l'interpréteur est figé en ROM.

## 1-2. LOGIQUE GENERALE DE L'OPTIMISATEUR.

Le principe même de l'optimisateur peut se résumer par la description de ses trois grandes étapes logiques (V. fig. 1-3):

A. la sélection recherche dans un programme (1) se trouvant dans FILE. BIBLI. NOOPTIM des séquences d'instructions identiques et choisit parmi les séquences détectées, celles dont la génération en mémoire MICALL fournira le gain maximum; elle construit les schémas (2) des séquences sélectionnées et produit deux fichiers :

- FILE. DESCRI. SQ. A. GEN qui contient les numéros des schémas et leurs descriptions;
- FILE. ADR. MODIF qui contient la correspondance entre
  - l'adresse d'une séquence sélectionnée;
  - le numéro du schéma correspondant;
  - les paramètres actuels décrits dans le même ordre que les paramètres formels du schéma correspondant.

Cette étape constitue la partie maîtresse de l'optimisateur; elle nous amènera à faire des choix qui risquent d'être déterminant quant à l'efficacité de l'interpréteur généré; c'est sur cette étape que nous concentrerons tout notre intérêt dans la suite de ce mémoire.

B. la génération traduit une liste de codes opérations se trouvant dans un schéma en un programme pour la mémoire MICALL (3); en fait, elle peut se résumer à remplacer (4) chaque code opération par les adresses des modules associés (V. paragr. 1-1) qui se trouvent dans la mémoire MIDEF, en tenant compte cependant :

- que certains modules ne sont plus nécessaires (5) ;
- qu'un arrangement adéquat des modules est à souhaiter (6) ;

---

(1) le langage dans lequel se trouvera ce programme sera défini en 1-3.

(2) la description précise du schéma sera donnée dans la suite de ce mémoire; signalons déjà qu'il contiendra : la liste des codes opérations des instructions, la description de la structure des branchements internes, la description de la répartition des paramètres formels.

(3) ce programme est en fait constitué des adresses des micro-programmes de la mémoire MIDEF que l'on veut appeler.

(4) le remplacement se fera en consultant une table.

(5) par exemple, certaines lectures mémoire ne seront plus nécessaires vu l'unification des paramètres répétés et la suppression de certains paramètres devenus implicites.

(6) notamment pour minimiser les temps de lecture mémoire qui ne se passent pas en simultanéité avec un traitement.

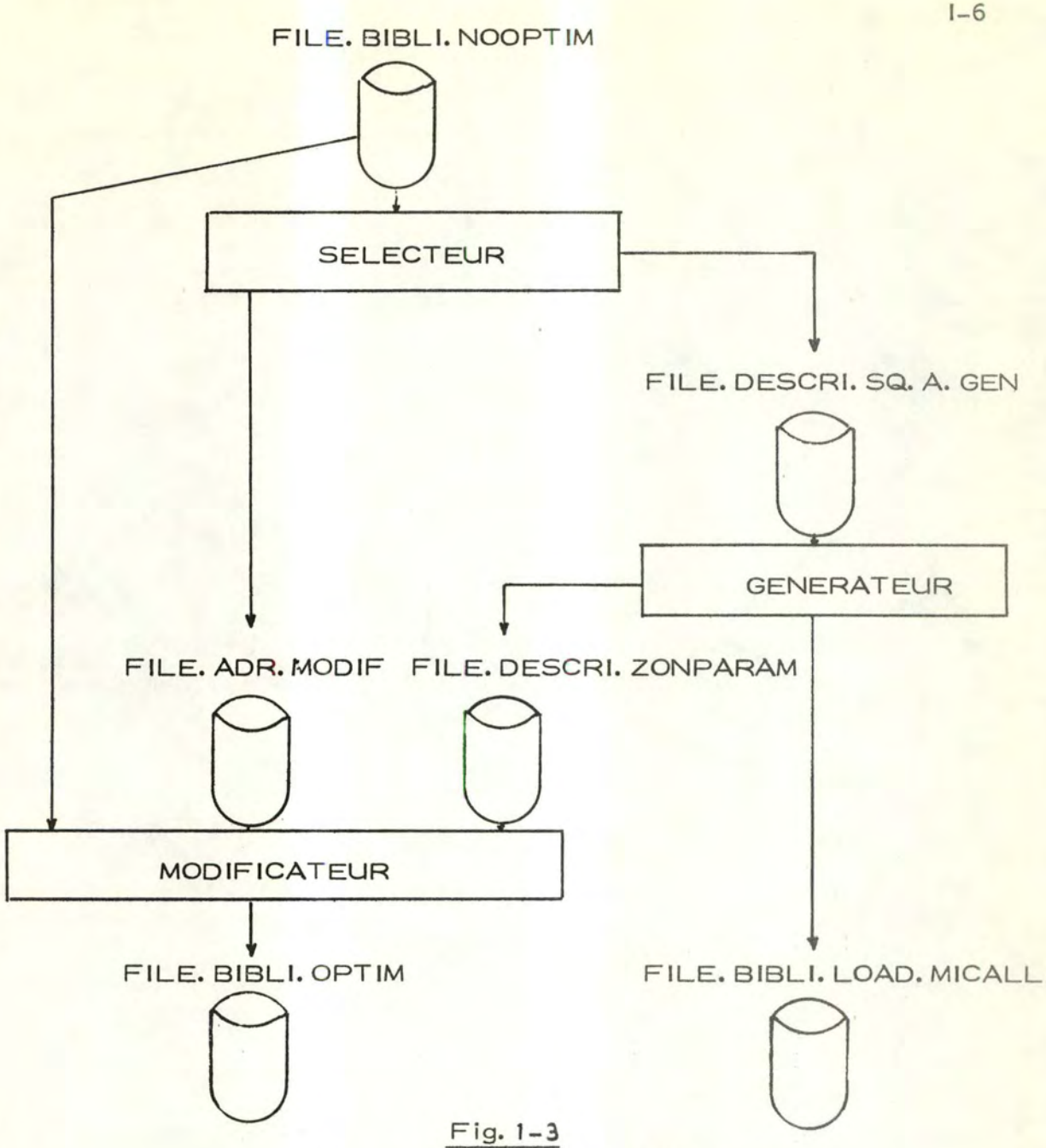



Fig. 1-3

Configuration de l'optimisateur

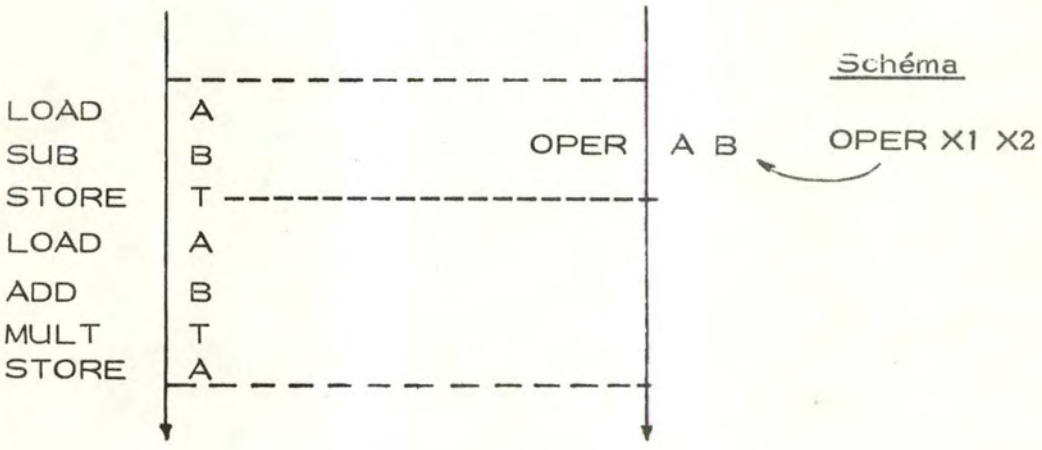



Fig. 1-4

Exemple de Modification pour la séquence représentant

(A-B)+(A+B) (Test une Variable temporaire)

La génération produit deux fichiers :

- FILE. BIBLI. LOAD. MICALL qui contient l'interpréteur à charger dans la mémoire MICALL lors de chaque exécution;
- FILE. DESCRI. ZONPARAM qui décrit le format de chaque nouvelle instruction, c'est-à-dire son code opération (1) et la position relative de chaque paramètre formel.

C. la modification remplace dans le programme origine (V. fig. 1-4) chaque séquence sélectionnée, par l'instruction machine créée lors de la génération et dans laquelle on a remplacé les paramètres formels par les paramètres actuels. La modification pose donc essentiellement un problème de suppression physique d'instructions à l'intérieur d'un programme, suivie du compactage de celui-ci; ce problème est simple lorsque toutes les adresses sont encore sous la forme symbolique; il se complique lorsque le programme est sous la forme absolue (adresses absolues) ou relogeable (adresses relatives) ; la seule solution consiste alors à garder des adresses symboliques dans le programme et à associer une table de correspondance entre les adresses symboliques et les adresses absolues ou relatives.

Une question se pose : 'quels gains pouvons-nous espérer du regroupement de plusieurs instructions machine en une nouvelle instruction machine plus complexe ? ' (2).
Nous les résumerons en :

- un gain en place mémoire utilisateur, vu que :
  - les codes opérations des instructions disparaissent;
  - les paramètres identiques fusionnent;
  - certains paramètres peuvent devenir implicites.
- le gain en temps de lecture mémoire associé; cependant ce gain n'est pas aussi grand que nous ne pourrions l'espérer; en effet, un grand nombre de lectures mémoire se déroulent en parallèle avec le processeur et ne freinent donc pas celui-ci; cependant l'autre partie constitue un temps mort pour le processeur.
  En diminuant le nombre de lectures mémoire nous pouvons espérer diminuer ces temps morts notamment en répartissant plus uniformément ces lectures.
- un gain en temps d'interface entre les instructions; il provient d'un gain sur les calculs d'adresses, sur les chargements de registres internes, sur le positionnement de certains codes conditions . . .

---

(1) ce code opération sera en relation directe avec l'adresse en mémoire MICALL.

(2) Attention ! ces gains sont à ajouter à ceux déjà obtenus grâce au mécanisme de la double interprétation associé à la hiérarchie des Mémoires.

## 1-3. RELATION ENTRE L'OPTIMISATEUR ET LES COMPILATEURS.

Comme nous l'avons dit au paragraphe 1-0, notre but n'est pas de redéfinir un nouveau software contenant une partie compilation et une partie optimisation, mais d'intégrer un OPTIMISATEUR dans une configuration existante.

Nous imposons en plus deux types de contraintes :

- contraintes de souplesse d'emploi de l'optimisateur :

  A. L'optimisateur ne peut perturber la configuration existante (1) ;

  B. Tous les langages de haut-niveau doivent pouvoir profiter de cette optimisation; à l'entrée de l'optimisateur, le programme doit donc se trouver sous une forme unique quelque soit le langage de haut-niveau de départ.

- contraintes en vue de faciliter la tâche de l'optimisateur :

  C. Le format d'entrée de l'optimisateur doit être facilement analysable ;

  D. Le format sera si possible sous une forme symbolique (pour faciliter la modification).

En vue de situer plus facilement l'optimisateur, nous décrirons brièvement la configuration actuelle (V. fig. 1-5); elle se compose d'un compilateur. Comme le décrit Gries (R26), un compilateur peut être divisé en deux grandes parties :

- l'ANALYSE dont le rôle est de découvrir la syntaxe et la sémantique du programme écrit en langage de haut-niveau et de construire le programme sous une forme interne (Triples, quadruples ...);
- la SYNTHESE qui, elle-même, peut être divisée en deux parties :
  - La PREPARATION qui produit une deuxième forme interne plus proche du langage machine, alloue la mémoire et optimise éventuellement le code;
  - La GENERATION qui traduit la deuxième forme interne en langage machine.

---

(1) en effet, l'emploi de l'optimisateur ne se révèle utile que lorsque le programme a été mis au point et est exploité couramment; il faut donc permettre aux utilisateurs, de continuer à compiler leurs programmes comme ils le faisaient avant.

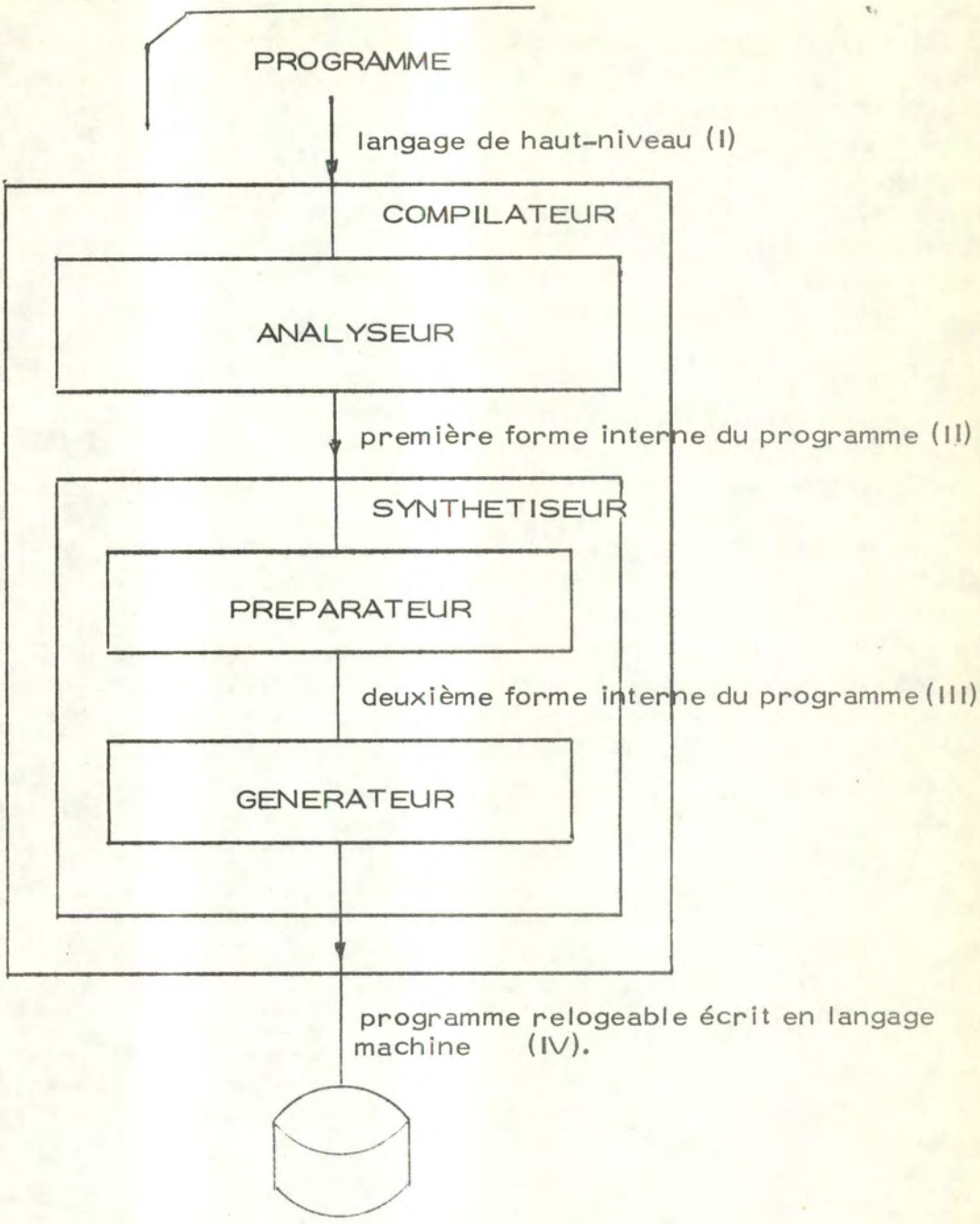



Fig. 1-5

Description des différentes représentations du programme dans un compilateur

La fig. 1-5 nous montre également les différents formats du programme au cours de la compilation; nous allons choisir, par élimination, un de ces formats comme entrée de l'optimisateur :

- en tenant compte du rôle particulier de l'analyseur et de la nécessité d'avoir à l'entrée de l'optimisateur un format unique pour tous les compilateurs (condition D, p. 1-8), nous ne pouvons choisir un format antérieur à la fin de l'analyse.

- le premier format qui s'offre donc à nous est le format II (V. fig. 1-5) (triples, quadruples ...); il satisfait bien les quatre conditions de la page 1-8 ; cependant dans la partie préparation du compilateur doivent encore se dérouler des optimisations (1) qui peuvent se résumer en des suppressions de code inutile (2); si l'on intercale avant cette partie, notre OPTIMISATEUR dont le but n'est pas de supprimer du code, mais d'accélérer du code existant en profitant de la technologie, on risque de garder du code inutile et de l'optimiser, ce qui est ridicule; le format II est donc à rejeter.

- dans le format III (V. fig. 1-5), le code intermédiaire a déjà été optimisé par le compilateur; pour autant que ce code existe (3) et soit standardisé au niveau de tous les compilateurs, il nous paraît être le code idéal; nous n'emploierons cependant pas ce format vu le peu de documentation que nous possédons sur les compilateurs Varian et la difficulté de s'introduire dans un programme existant lorsque tous les interfaces ne sont pas bien définis.

- dans le cas de l'ordinateur Varian, le format IV (V. fig. 1-5), c'est-à-dire le code chargeable, nous a semblé beaucoup trop lourd pour être manipulé efficacement ; sa complexité provient du fait que le chargeur contribue à la deuxième passe de l'Assembleur.

- finalement, nous avons choisi le langage machine absolu en nous rappelant qu'il ne vérifie pas la condition D p. 1-8, ce qui compliquera le modificateur.

---

(1) ces optimisations sont plus ou moins élaborées selon le compilateur.

(2) Exemple : Load suivi de Store.

(3) rien en nous dit que tous les compilateurs respectent exactement le modèle de Gries.

## 1-4. PROBLEMES DE FIABILITE.

Nous signalerons ici quelques problèmes de fiabilité que nous avons détectés; nous ne les détaillerons pas car leur étude pourrait à elle seule faire l'objet d'un travail.

Le problème peut se poser à deux niveaux :

- Niveau Software : il faut en effet se rappeler que notre COMPILATEUR/OPTIMISATEUR joue un rôle de protection dans le système en empêchant l'utilisateur de micro-programmer (et de devenir ainsi le maître de la machine).
  Si un micro-programme généré est faux ou si un appel à un micro-programme est faux toute la protection risque de tomber et l'on ne peut savoir quel traitement va être effectué, ni quelles données vont être modifiées.
  Il est donc très important d'avoir un software fiable.

- Niveau Firmware : une protection au niveau firmware est nécessaire :

  - contre les fautes du programme se trouvant en mémoire MICALL (par exemple, une adresse fausse d'un module de la mémoire MIDEF) (1);

  - contre les fautes du programme se trouvant en mémoire utilisateur (mauvais code opération ou mauvais paramètres) (2).

Cette protection pourrait être assurée par :

- un mécanisme d'indirection lors de chaque appel (comme c'est le cas dans la plupart des firmware actuels);

- des tests de validité dans les modules de la mémoire MIDEF.

Nous n'entrerons pas dans les détails de cette protection.

---

(1) ces fautes peuvent provenir de fautes software (COMPILATEUR/OPTIMISATEUR).

(2) ces fautes peuvent provenir de fautes software (COMPILATEUR/OPTIMISATEUR) ou d'erreurs lors de l'exécution du programme (exécution de données).

# CHAPITRE 2

## DESCRIPTION LOGIQUE DU 'SELECTEUR'.

2-0. Jusqu'à présent, nous avons situé notre application : le SELECTEUR,

- d'abord parmi les grands courants de recherche (Ch. 0);
- ensuite dans un software décrit sur une machine donnée, l'ordinateur VARIAN 72 (Ch. 1)

Nous allons maintenant étudier cette application à un niveau logique (1) ;

- nous commencerons en 2-1 par :
  - introduire les grandes étapes logiques de la SELECTION;
  - justifier leur présence;
  - préciser le type de méthodologie employée dans chacune d'elles.
- nous détaillerons ensuite la logique de chacune d'elles (2-2, 2-3, 2-4).

---

(1) l'implantation physique sera décrite au chapitre 3.

## 2-1. LES GRANDES ETAPES LOGIQUES.

En résumant ce que nous avons déjà dit au sujet du SELECTEUR, nous pouvons énoncer le problème qu'il résout de la manière suivante; 'étant donné un programme écrit en langage machine sous forme absolue, découvrir dans ce programme, les séquences d'instructions qu'il est préférable de générer en mémoire MICALL pour avoir le rendement maximum; rechercher la structure de ces séquences.'

Comme nous le montre la fig. 2-1, le SELECTEUR peut être divisé en deux grandes parties logiques :

- la PREPARATION dont les deux buts sont les suivants :
  - mettre le programme sous la forme la mieux adaptée pour la recherche des séquences d'instructions ;
  - fournir des renseignements supplémentaires concernant la structure du programme;
- le REPERAGE et le CHOIX des séquences qui sont les plus aptes a être générées en mémoire MICALL.

La PREPARATION peut elle-même être subdivisée en deux phases :

- la PHASE PRELIMINAIRE de PSEUDO-CHARGEMENT (V. fig. 2-1) qui traduit le programme chargeable se trouvant dans FILE.BIBLI.NOOPTIM en un programme absolu stocké dans FILE. MOD. CHARG; cette traduction , rendue nécessaire par le choix d'un programme absolu comme entrée du SELECTEUR, peut être réalisée par le chargeur standard de l'ordinateur VARIAN 72;
- la PHASE 1 d'ANALYSE DES STRUCTURES DU PROGRAMME (V. fig. 2-1) qui construit à partir du programme absolu se trouvant dans FILE. MOD. CHARG et de renseignements supplémentaires (1) se trouvant dans FILE. BIBLI. NOOPTIM :
  - la liste des codes opérations des instructions du programme (programme décomposé);
  - un fichier FILE. GRAPH contenant la structure des branchements et les paramètres des instructions.

Si l'on se rappelle que, dans un programme sous forme absolue, une donnée ne peut être différenciée d'une instruction tant que l'on ne connaît pas la structure des branchements, cette phase semble indispensable avant toute analyse des instructions du programme.

---

(1) les renseignements concernent les références externes (EXTRN) et les entrées (ENTRY) du programme.

Cette phase utilise la théorie des graphes, non seulement pour représenter la structure du programme (1), mais aussi pour découvrir certaines propriétés de cette structure.

La deuxième partie, REPERAGE et CHOIX, est constituée, elle aussi de deux phases indépendantes :

- la PHASE 2 de REPERAGE DES SEQUENCES IMPLEMENTABLES (V. fig. 2-1) qui repère des séquences d'instructions, regroupe celles qui ont la même structure et élimine celles qui ne sont pas implémentables (2); elle génère deux fichiers :
  - FILE. SITUE. CONTEXT qui décrit les séquences dans leur contexte (3);
  - FILE. DESCRI. SQ qui décrit les schémas des séquences (4).

  Nous emploierons la théorie des ensembles en vue de visualiser le classement.

- la PHASE 3 de CHOIX DES SEQUENCES A IMPLEMENTER (V. fig. 2-1) qui commence par évaluer les gains individuels qu'apporterait chaque schéma de séquences si on l'implémentait en mémoire MICALL; elle résout ensuite le problème suivant : étant donné un ensemble de schémas, lesquels vais-je générer en mémoire MICALL pour avoir le gain maximum, en tenant compte cependant que la place en mémoire MICALL est limitée; en fonction des résultats obtenus, elle met à jour les fichiers FILE. SITUE. CONTEXT et FILE. DESCRI. SQ pour en faire des fichiers FILE. ADR. MODIF. et FILE. DESCRI. SQ. A. IMPL.
  Cette phase fera largement appel à la programmation linéaire .

---

(1) la correspondance exacte entre la structure du programme et le graphe sera faite au paragraphe 2-2 -1.

(2) ce point sera précisé au paragraphe 2-3.

(3) c'est-à-dire leur adresse, leur longueur et les paramètres actuels...

(4) une notion intuitive des schémas de séquences a déjà été donnée, nous la préciserons au paragraphe 2-3.

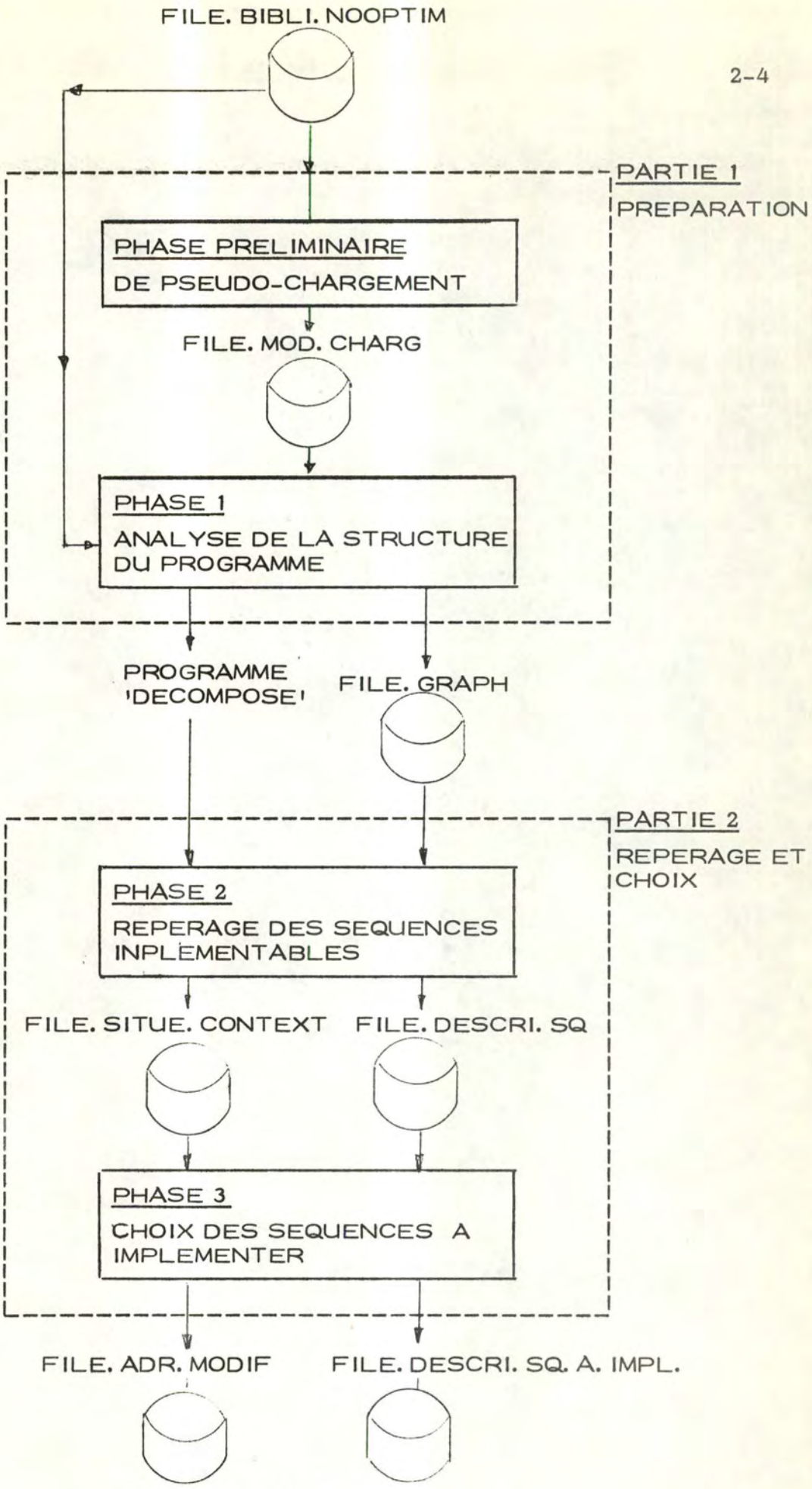



Fig. 2-1

Description logique générale du SELECTEUR

## 2-2. PHASE 1 : ANALYSE DES STRUCTURES LOGIQUES DU PROGRAMME.

**2-2-0.** Comme nous l'avons déjà dit, la théorie des graphes constitue la base de cette phase.
Dans ce paragraphe nous nous intéresserons tout d'abord à la correspondance exacte entre la structure du programme et sa représentation sous forme de graphe (2-2-1).
Nous porterons ensuite notre attention sur les deux grandes parties qui constituent cette phase (V. fig. 2-2), c'est-à-dire :

- la création du graphe du programme (2-2-2);
- la détection des circuits élémentaires dans le graphe (2-2-3).

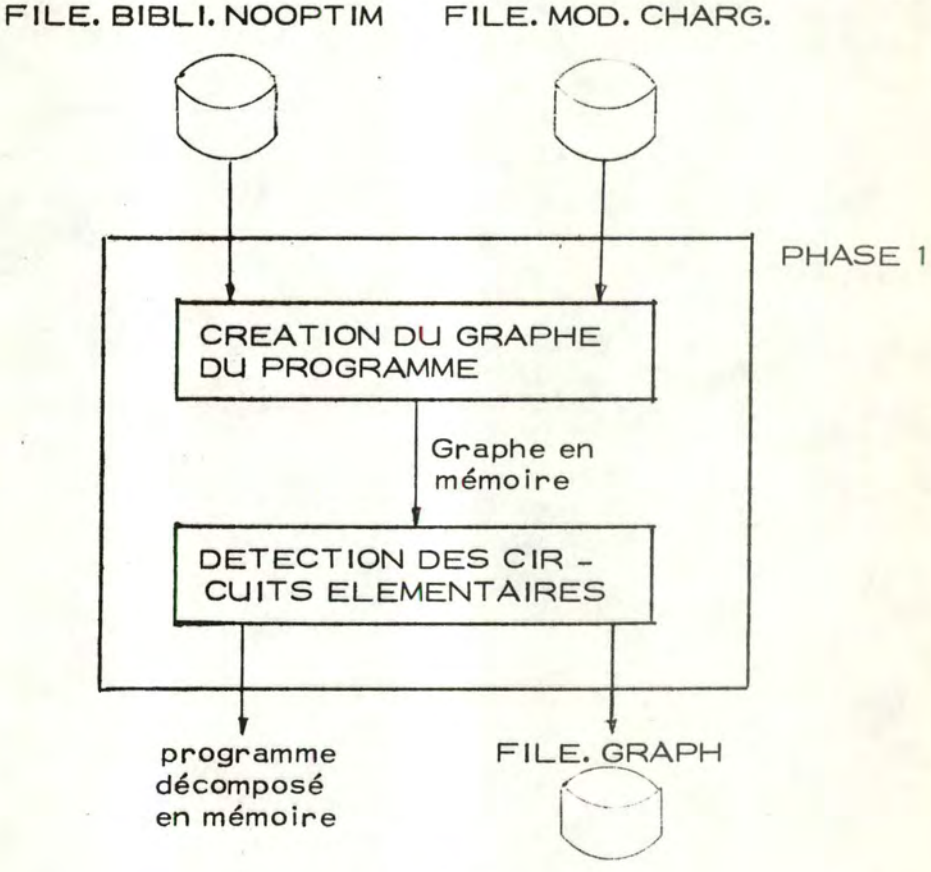



Fig. 2-2

## 2-2-1. STRUCTURE DU PROGRAMME ET REPRESENTATION SOUS FORME DE GRAPHE.

Nous appelons 'structure du programme', l'ensemble des branchements et des références de ce programme et les différentes liaisons qui existent entre ces éléments.
La correspondance entre la structure d'un programme et sa représentation sous forme de graphe semble donc évidente; la fig. 2-3 nous montre les principaux cas de cette correspondance sur un programme schématisé.
La correspondance peut se résumer comme suit :

1. à chaque branchement et à chaque référence du programme est associé un sommet du graphe; cependant à un branchement référencé n'est associé qu'un seul sommet;

2. un sommet initial est généré et correspond au début du programme;

3. à chaque 'flèche de branchement' et à chaque séquence d'instructions non interrompue (1) par un branchement ou une référence est associé un arc orienté du graphe;

4. un sommet supplémentaire (et les arcs orientés correspondants) est associé à chaque appel d'un module externe et représente le corps de ce module vu à partir du programme (2).

---

(1) la première instruction d'une telle séquence peut être référencée.

(2) la ou les fins de programmes sont assimilées à des branchements externes sans retour au programme.

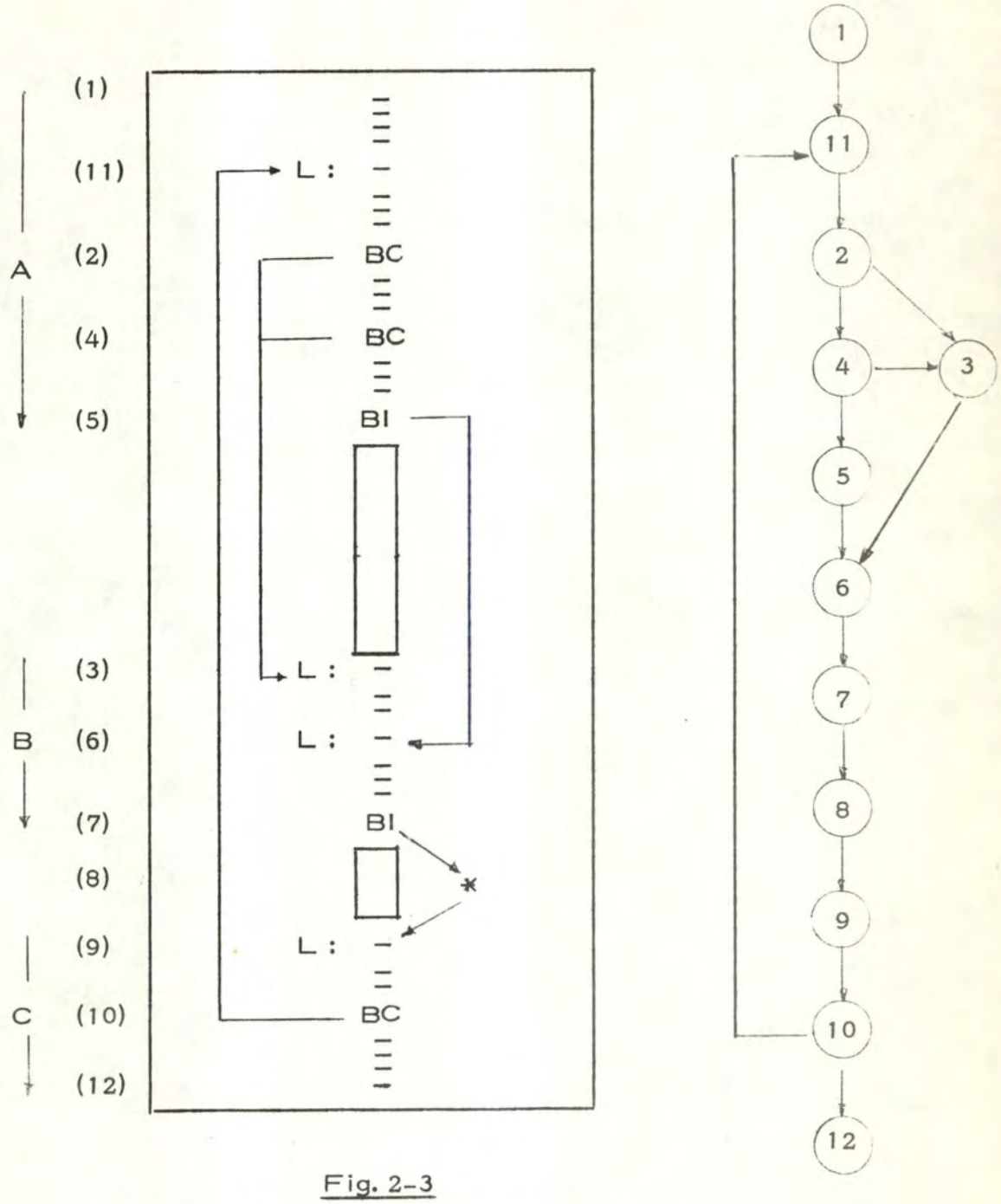



Fig. 2-3

Correspondance entre la structure d'un programme et sa représentation sous forme de graphe.

Légende : - instruction normale.
BC Branchement conditionnel
BI Branchement inconditionnel
L : référence
* module externe
zone blanche = zone de données.

Les chiffres entre parenthèses marquent l'ordre de construction des sommets du graphe; les lettres marquent l'ordre de parcours du programme.

## 2-2-2. CREATION DU GRAPHE DU PROGRAMME.

En vue de détecter la structure d'un programme, nous décrirons une méthode dérivée du principe d'exécution.
Elle consiste en un balayage systématique du programme à partir de l'entrée principale, en tenant compte à chaque pas de la longueur de l'instruction en cours de sa nature; dans le cas où l'instruction en cours est un branchement, chacune des directions est parcourue successivement (1), ce qui nous permet d'examiner chaque instruction une et une seule fois, contrairement à ce qui se passe généralement lors d'une exécution.

L'algorithme, simple en apparence, pose tout de même deux problèmes :

- lorsque l'on appelle un module externe au programme, le branchement vers ce module peut être détecté, par contre, le ou les retours ne pourraient être connus qu'en analysant le module appelé, ce qui n'est pas notre but;

- l'algorithme suppose la connaissance de toutes les adresses de branchements sous une forme absolue; or, si l'on regarde les différents branchements qu'offre le jeu d'instructions VARIAN 72 (V. fig. 2-4), on constate que dans le cas du saut indexé (IJMP), l'adresse est le résultat d'un calcul dépendant plus ou moins directement des données et ne peut donc être connue qu'au moment de l'exécution.

En vue de résoudre ces deux problèmes, nous commencerons par analyser la manière dont est réalisé par un compilateur VARIAN 72 l'appel à un module externe (V. fig. 2-5).
L'appel proprement dit consiste en un branchement avec stockage de l'adresse suivante(JMPM) dans une zone mémoire (ADR) ; ce branchement est généralement suivi de quelques paramètres (Pi).
L'appel des paramètres et le branchement de retour se font au moyen d'adresses indirectes postindexées (ADR * + dpp) ; le saut de retour est donc un saut indexé (IJMP).
L'appel d'un module externe pose donc simultanément le problème du saut indexé et le problème de l'adresse de retour d'un module externe, mais ceux-ci peuvent être facilement résolus si l'on se rappelle :

- que les déplacements (dpp 2, dpp 4 ...) sont connus par le compilateur (2) et
- que seul le programme principal est analysé.

---

(1) sauf si certaines directions ont déjà été parcourues.

(2) c'est le rôle des renseignements supplémentaires que nous avons introduits au paragraphe 2-1.

Le problème du saut indexé n'est cependant pas entièrement résolu; le saut indexé pourrait être employé à d'autres usages, à l'intérieur du programme; il ne semble pas que ce soit le cas dans le code généré par les compilateurs VARIAN; de toute façon nous prenons l'option de bloquer la construction du graphe si de tels cas se présentaient.

Avant d'approfondir l'algorithme de construction du graphe, nous donnerons une définition :

A un moment donné de l'algorithme, nous appellerons 'entrée du programme' une référence déjà détectée, mais non encore explorée; au départ nous posséderons donc une seule entrée appelée 'entrée principale' et, à chaque branchement, l'algorithme créera de nouvelles entrées.

Fig. 2-4 : Types de branchements du VARIAN 72.

JMP : Jump; saut conditionnel ou inconditionnel à une adresse absolue.

JMPM : Jump and mark; saut conditionnel ou inconditionnel à une adresse absolue avec rangement de l'adresse suivante.

SRE : Skip if register equal; saut conditionnel deux mots plus loin.

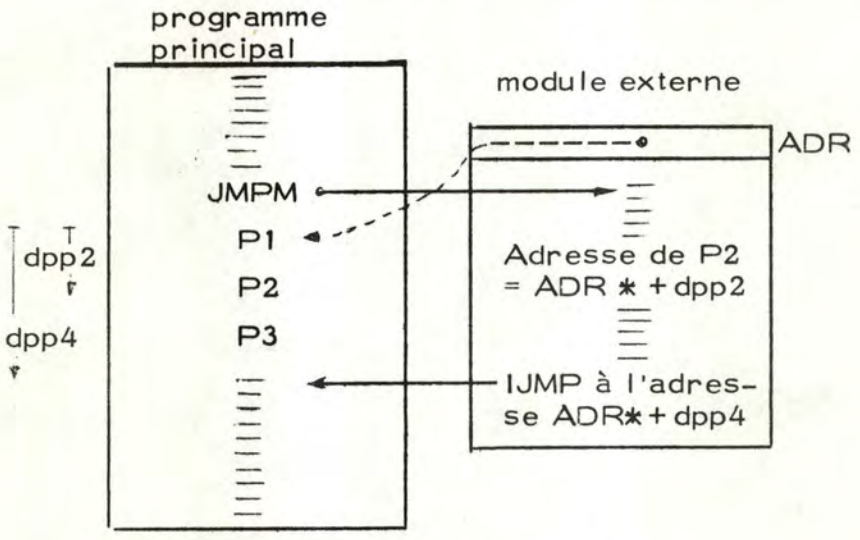
BT : Bit test; saut conditionnel à une adresse absolue.

JSR : Jump and return address; saut inconditionnel à une adresse absolue et stockage de l'adresse suivante dans un registre.

IJMP : Indexed jump; saut inconditionnel à une adresse absolue indexée par le contenu d'un registre.

Remarque : tous ces 'jump' permettent des indirections.

Fig. 2-5 : Appel à un module externe.




Légende : - une instruction normale
P1, P2, P3 = paramètres à passer au module externe
dpp2, dpp4 = déplacements connus à la compilation du module externe et du programme principal.
ADR = adresse de la zone où l'on va stocker l'adresse suivante.
ADR* adresse indirecte.

Nous pouvons maintenant détailler l'algorithme :

A : s'il reste encore des entrées du programme,
  alors - choisir une de ces entrées (1);
  - lire la première instruction se trouvant à cette entrée;
  - aller en B;
  sinon fin de l'algorithme.

B : si l'instruction en cours est référencée par un 'label' déjà découvert par l'algorithme,
  alors si l'instruction référencée a déjà été explorée,
  alors aller en A ;
  sinon aller en C;
  sinon aller en C .

C : si l'instruction en cours est un branchement,
  alors si le branchement est un IJMP,
  alors fin de l'algorithme ;
  sinon - créer les deux sommets du graphe correspondant à l'origine et à la destination du branchement si ces sommets n'existent pas déjà (3);
  - créer les arcs orientés associés;
  - si le branchement est inconditionnel,
  alors aller en A ;
  sinon aller en D;
  sinon aller en D.

D : - lire l'instruction suivante en séquence (2) ;
  - aller en B.

Remarque :

Dans la fig. 2-3, les chiffres entre parenthèses montrent l'ordre de création des sommets du graphe sur un exemple schématisé.

---

(1) logiquement il n'y a pas de critères de choix, physiquement, il sera lié à une optimisation de lectures sur disque (V. ch. 3).

(2) cela suppose un calcul de la longueur de l'instruction en cours.

(3) la destination du branchement peut être une référence non encore explorée, c'est-à-dire une nouvelle entrée du programme.

## 2-2-3. DETECTION DES CIRCUITS ELEMENTAIRES DU GRAPHE.

Cette deuxième partie de la PHASE2 recherche les circuits élémentaires (1) du graphe (2) et leurs niveaux d'imbrication (3).

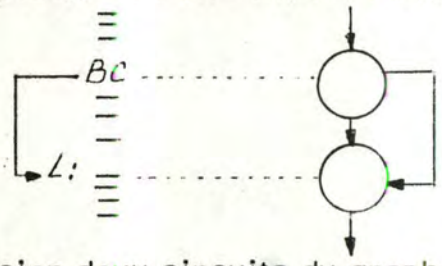
Plusieurs algorithmes existent pour détecter les circuits élémentaires d'un graphe (4); pour différentes raisons, ils sont généralement mal adaptés à notre application :

- la plupart emploient une représentation (5) matricielle du graphe (6); or, dans notre application, nous avons choisi (v. paragraphe 3-1-3) une représentation sommet par sommet (7);
- l'adaptation de ces algorithmes à certaines particularités de notre graphe semble difficile.

---

(1) un circuit élémentaire est un circuit qui ne contient pas deux fois le même sommet; il représente en fait une boucle du programme; la nécessité de la détection de ces circuits sera évoquée dans la suite de ce mémoire.

(2) en réalité, le graphe comme nous l'avons défini est un 2-graphe (c'est-à-dire qu'un sommet est relié à un autre sommet par deux arcs au plus); on s'en convaincra grâce à la figure suivante :




Différencier deux circuits du graphe pour de tels cas ne nous intéresse pas; aussi négligerons-nous un des deux arcs dans l'algorithme de détection des circuits.

(3) le niveau d'imbrication d'une boucle est le nombre de boucles qui la contiennent; le calcul de ce nombre ne pose pas de difficultés.

(4) (R31) méthode de la multiplication de la matrice booléenne, méthode de la multiplication latine, méthode des approximations successives ..

(5) cette raison tient compte de l'implémentation physique du graphe.

(6) c'est-à-dire une matrice nxn (où n est le nombre de sommets du graphe) et dans laquelle un élément spécifie si un arc du graphe existe.

(7) la représentation sommet par sommet fait correspondre à chaque sommet la liste de ses suivants.

L'algorithme choisi dans notre application s'inspire (1) de l'algorithme E. C. de TIERNAN (R33); le principe de base de celui-ci est relativement simple et naturel; parallèlement à son développement, le lecteur pourra suivre pas à pas l'évolution sur un exemple (v. fig. 2-6 a, b, c).

Au départ, les sommets doivent être numérotés (2). On considère alors un à un, chaque sommet dans l'ordre de la numérotation (étape A p. 2-15 ) et pour chacun d'eux on recherche tous les circuits élémentaires possibles, qui ne contiennent pas de sommets 'inférieurs' (3) au sommet de départ (étapes B, C, D, E p. 2-15).
La recherche des circuits correspondant à un sommet de départ se fait par extension pas à pas d'un chemin à partir de celui-ci; à chaque pas, on rajoute au chemin un sommet suivant (4) du dernier sommet de ce chemin qui vérifie 3 conditions (étape C p 2-15); elles ont respectivement pour but :

- de construire un chemin élémentaire;
- de ne pas construire un chemin déjà construit au début de l'algorithme;
- de ne considérer chaque circuit qu'une seule fois (5).

Si l'on ne parvient plus à trouver un tel sommet, on regarde si le sommet de départ du chemin est un suivant du dernier sommet de ce chemin; dans ce cas, on a découvert un circuit.

S'il ne reste plus qu'un seul sommet dans le chemin, on choisit un nouveau sommet de départ.
Sinon, on enlève le dernier sommet du chemin et on essaye d'étendre le nouveau chemin dans une autre direction (étape E p. 2-15).

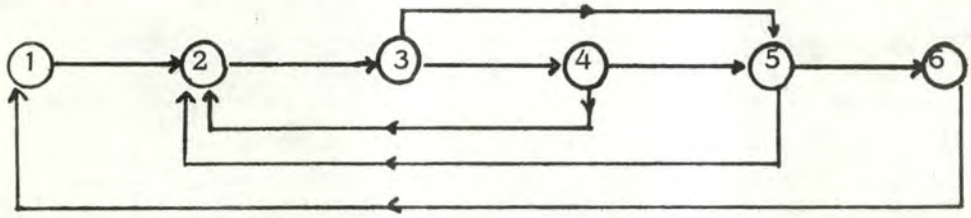



fig. 2-6-a.

---

(1) Les modifications qui y sont apportées pour tenir compte des particularités du graphe, seront décrites ci-après.

(2) L'ordre de numérotation importe peu.

(3) La relation d'ordre est induite par la numérotation.

(4) Un sommet suivant dans le graphe.

(5) Le point ne pourra être pleinement compris qu'après l'explication de la détection des circuits; ce que nous ferons ci-après.

| Sommets | Liste des sommets suivants |
|---|---|
| 1 | 2 |
| 2 | 3 |
| 3 | 4, 5 |
| 4 | 2, 5 |
| 5 | 2, 6 |
| 6 | 1 |

fig. 2-6-b : description du graphe.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | | 2 | | 3 | | 4 | |
| 1, 2 | | 2, 3 | | 3, 4 | | 4, 5 | |
| 1, 2, 3 | | 2, 3, 4 | | 3, 4, 5 | | 4, 5, 6 | |
| 1, 2, 3, 4 | | 2, 3, 4, 5 | | 3, 4, 5, 6 | | 4, 5 | |
| 1, 2, 3, 4, 5 | | 2, 3, 4, 5, 6 | | 3, 4, 5 | | 4 | |
| 1, 2, 3, 4, 5, 6 | B | 2, 3, 4, 5 | B | 3, 4 | | | |
| 1, 2, 3, 4, 5 | | 2, 3, 4 | B | 3 | | 5 | |
| 1, 2, 3, 4 | | 2, 3 | | 3, 5 | | 5, 6 | |
| 1, 2, 3 | | 2, 3, 5 | | 3, 5, 6 | | 5 | |
| 1, 2, 3, 5 | | 2, 3, 5, 6 | | 3, 5 | | | |
| 1, 2, 3, 5, 6 | B | 2, 3, 5 | B | 3 | | 6 | |
| 1, 2, 3, 5 | | 2, 3 | | | | | |
| 1, 2, 3 | | 2 | | | | | |
| 1, 2 | | | | | | | |
| 1 | | | | | | | |

Légende : B signifie qu'une boucle est détectée.

fig. 2-6-c : description de l'évolution du chemin élémentaire.

A : pour chaque sommet du graphe pris dans l'ordre croissant de la numérotation,

B : considérer ce sommet comme sommet de départ d'un chemin élémentaire.

C : rechercher dans le graphe un sommet suivant du dernier sommet du chemin élémentaire, qui vérifie 3 conditions :

1. il n'appartient pas encore au chemin ;
2. il n'est pas pour le moment 'fermé' à l'extension ; (1)
3. il est 'plus grand' que le sommet de départ du chemin.

D : si l'on a trouvé un tel sommet,

alors étendre le chemin avec le sommet trouvé;
aller en C ;

sinon si le sommet de départ est un suivant du dernier sommet du chemin élémentaire,

alors (on a un circuit); aller en E;

sinon aller E.

E : si le chemin élémentaire ne contient qu'un sommet,

alors (tous les circuits contenant ce sommet ont été considérés); aller en A (pour prendre le sommet suivant);

sinon enlever le dernier sommet du chemin élémentaire;
aller en C .

à la fin fin de l'algorithme.

---

(1) Un sommet 'fermé à l'extension' à un moment de l'algorithme correspond à une 'direction' dans laquelle le chemin actuel a déjà été étendu; par exemple (v. fig. 2-6-c) à la 7ème ligne de la première colonne, le sommet 6 est 'fermé à l'extension', à la 13ème, les sommets 4 et 5 sont 'fermés à l'extension'.

En déhors des améliorations que nous avons apportées au niveau de l'implantation (1), nous avons pris une option importante au niveau logique pour tenir compte d'une particularité du graphe. Nous restreignons l'ensemble des sommets de départ de l'extension aux sommets représentant des 'labels' référencés par des branchements arrières (2); il est en effet évident que toutes les boucles d'un programme doivent au moins avoir un branchement arrière. En dehors du fait que l'algorithme choisi est théoriquement le plus efficace (R33) pour un graphe quelconque, cette petite rajoute risque de nous faire gagner un temps considérable (3).

Une autre option a été prise quant à la définition de la numérotation des sommets; nous la définirons implicitement par l'ordre dans lequel se trouvent les instructions correspondant à ces sommets dans le programme.

---

(1) TIERNAN présente l'algorithme en utilisant trois matrices (R33) :

- G (nxn) dont chaque ligne représente l'ensemble des sommets suivant du sommet correspondant; nous transformerons chaque ligne de la matrice en une liste, ce qui ne pose aucun problème au niveau de l'algorithme;
- H (nxn) dont chaque ligne contient à un moment donné tous les sommets qui sont fermés à l'extension à partir du sommet correspondant à la ligne; nous remplacerons cette matrice par un pointeur dans chacune des chaînes du graphe et nous prendrons pour convention que seuls les sommets décrits à droite de ce pointeur sont fermés à l'extension;
- P (n) qui contient le chemin élémentaire et joue en fait le rôle d'un 'stack'.

(2) Ceux-ci peuvent être détectés facilement lors de la création du graphe.

(3) Dans l'exemple, les colonnes 3 et 4 n'existent plus (v. fig. 2-6-c).

## 2-3. PHASE 2 : REPERAGE DES SEQUENCES IMPLEMENTABLES ET CONSTRUCTION DES SCHEMAS.

2-3-0. Comme nous le montre la fig. 2- 7 cette phase peut être divisée en deux parties logiques qui s'appellent mutuellement; chaque séquence est d'abord repérée dans le programme, puis elle est classée (1); cependant ces deux parties ne sont pas aussi bien séparées qu'elles ne le paraissent (2); les séquences sont en effet repérées dans un certain ordre, et le repérage contribue donc à la première étape du classement.

Dans les paragraphes suivants,

- nous commencerons par exposer le principe de repérage des séquences (2-3-1);
- nous étudierons ensuite les différentes sortes de classements et leur nécessité (2-3-2);
- nous décrirons enfin les algorithmes de repérage et de classement (2-3-3).

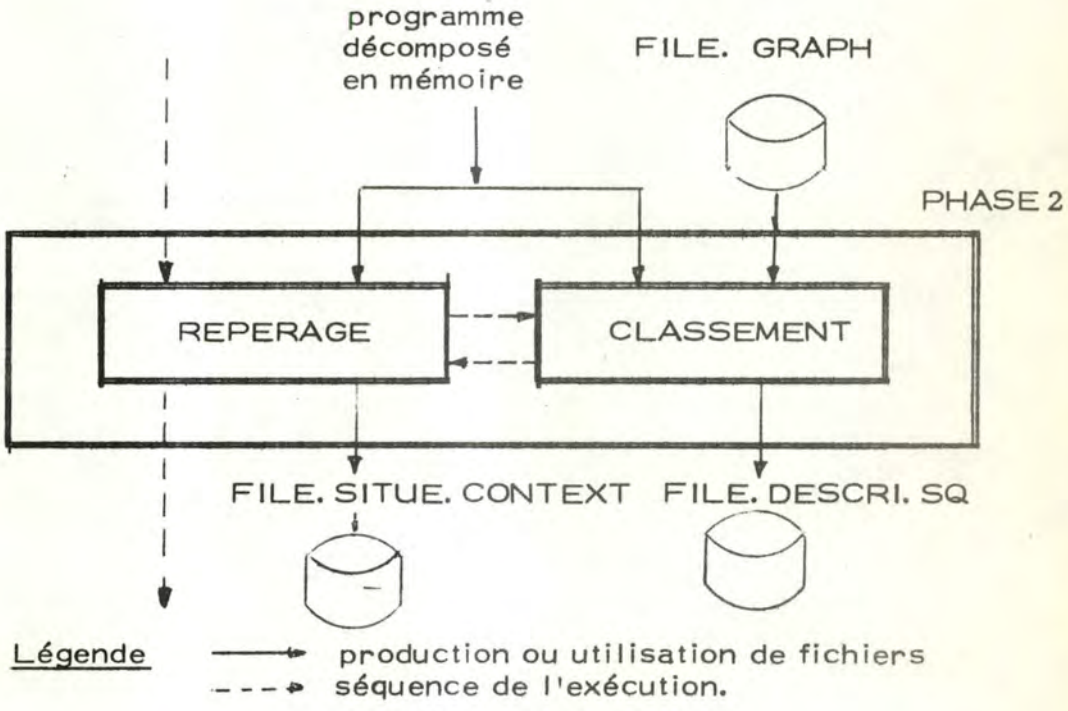



Fig. 2- 7

---

(1) le rôle exact du classement sera précisé au paragraphe 2-3-2.

(2) nous détaillerons ce point au paragraphe 2-3-3.

## 2-3-1. PRINCIPE DE REPERAGE DES SEQUENCES.

Jusqu'à présent, la séquence correspondait pour nous, à la notion intuitive de 'suite d'instructions'; nous ne nous étions pas encore occupés de la manière dont était repérée cette suite dans un programme.

Plusieurs solutions sont possibles; dans notre application, les instructions sont prises dans l'ordre décrit dans le programme (1). On associe à chaque instruction d'un programme une famille de séquences ayant la même origine (2); les séquences de cette famille s'obtiennent par extension pas à pas à partir de l'instruction origine (V. fig. 2- 8) en suivant l'ordre de description des instructions dans le programme.
L'union de toutes les familles d'un programme constitue l'ensemble des séquences du programme ( V. fig. 2-9 )

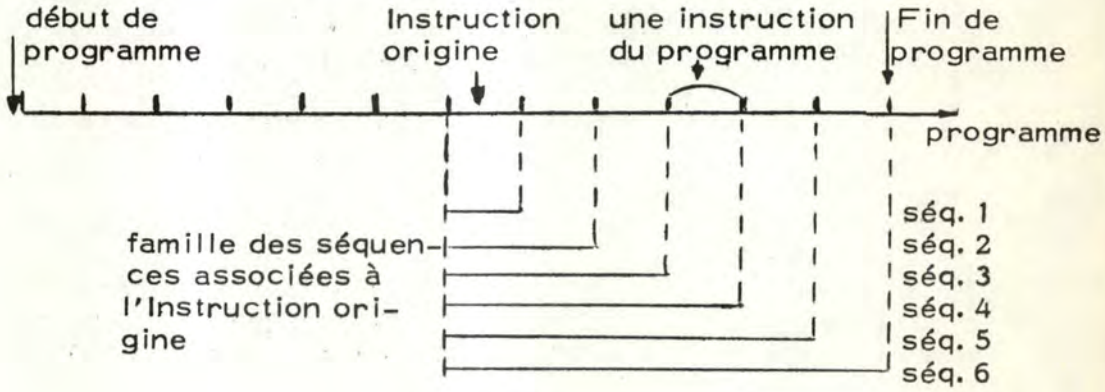
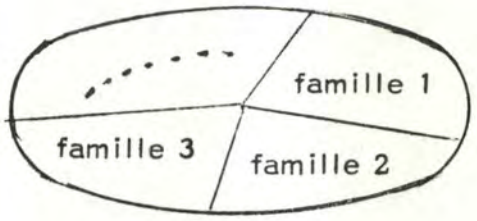



Fig. 2- 8

Famille des séquences associées à l'instruction origine.




Ensemble des séquences du programme

Fig. 2-9

---

(1) une autre solution consisterait à prendre les instructions dans l'ordre décrit par le programme, c'est-à-dire en tenant compte de la structure du programme (branchements); cette solution permettrait en principe de découvrir plus de séquences mais présenterait en revanche de sérieuses difficultés d'implémentation et risquerait de donner un algorithme lourd et peu efficace.

(2) cette instruction est l'origine.

## 2-3-2. NECESSITE ET DESCRIPTION DES DIFFERENTES SORTES DE CLASSEMENTS.

Deux types de critères assurent le classement de l'ensemble des séquences d'un programme :

1. Les premiers sont des critères de regroupement ; leur emploi se justifie aisément; en effet, bien que l'implémentation d'une séquence particulière nous permette de générer en mémoire MICALL le programme le plus efficace (1), la contribution de cette implémentation au niveau du programme reste généralement faible car ce gain n'est pas répété ; on peut donc penser que la mémoire MICALL est mal utilisée (2).

   La nécessité de considérer des séquences répétées (3) s'impose donc, or en pratique, les séquences identiques au point de vue codes opérations et paramètres sont rares; par contre dans les codes générés par les compilateurs, on rencontre fréquemment des séquences possédant la même suite de codes opérations et la même structure interne de branchements; de telles séquences peuvent avoir la même implémentation en mémoire MICALL à condition que l'on puisse décrire leurs paramètres de manière unique; il est donc normal de la regrouper.

   Ce regroupement peut être vu comme une suite de deux partitionnements (V. fig. 2-10) :

   - le partitionnement de l'ensemble des séquences d'un programme; un élément de cette partition s'appelle un type de séquences et contient toutes les séquences possédant la même suite de codes opérations;

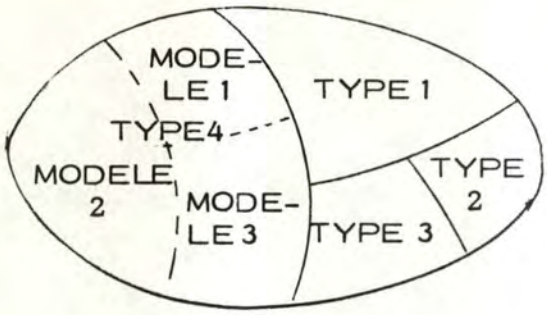
   - le partitionnement de chaque type de séquences; un élément de ces partitions s'appelle un modèle de séquences et contient toutes les séquences d'un même type qui possèdent la même structure de branchements internes.

---

(1) cette efficacité provient du fait que l'on peut tenir compte de toutes les particularités de la séquence pour l'implémenter; ainsi, tous les paramètres peuvent devenir internes, ce qui entraîne un gain de place mémoire utilisateur lors de l'appel et un gain de temps probable.

(2) notre but est, en effet, d'implémenter en mémoire MICALL les traitements fournissant 'la densité de gain maximum (au niveau du programme) par unité de mémoire MICALL utilisée'.

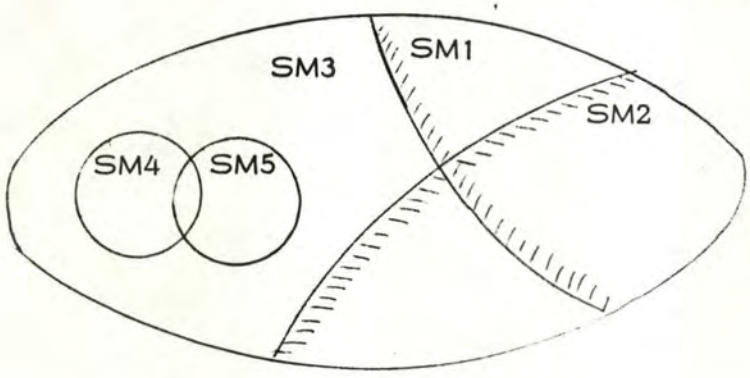
(3) deux sortes de répétitions existent :
- Répétitions statiques, c'est-à-dire des répétitions dans la description du programme; elles fournissent à la fois une augmentation du gain en temps d'exécution et en place mémoire utilisateur;
- Répétitions dynamiques, c'est-à-dire provoquées à l'exécution par les boucles du programme; elles n'augmentent que le gain en temps d'exécution.




Ensemble des séquences d'un programme

Fig. 2-10

D'après ce que nous venons de dire, toutes les séquences appartenant à un même 'sous-ensemble d'un modèle'(appelé sous-modèle (1) : V. fig. 2-11) peuvent jouir de la même représentation en mémoire MICALL si l'on peut uniformiser leurs paramètres.




SMO = MODELE

(SM signifie sous-modèle)

Fig. 2-11

Pour mieux situer le problème de l'uniformisation des paramètres nous nous baserons sur les quatre séquences décrites à la fig. 2-12-a (2); nous résumons l'uniformisation des paramètres des séquences d'un sous-modèle, par les deux règles suivantes :

A : si dans toutes les séquences d'un sous-modèle, la même instruction a les mêmes paramètres,
alors ce paramètre devient interne (3);
sinon ce paramètre devient un paramètre externe et est représenté dans les caractéristiques du sous-modèle par un paramètre formel (noté Pi) (4).

---

(1) en particulier, le modèle lui-même est un sous-modèle.
(2) elles appartiennent à un même type et à un même modèle dont nous donnons les caractéristiques à la fig. 2-12-b (1ère et 2ème colonne).
(3) exemple : paramètre T de la cinquième instruction des séquences du sous-modèle 1 (V. fig. 2-12-a et b)
(4) exemple : paramètre de la première instruction qui a pour valeur A, W, L, B dans les séquences du sous-modèle 1 (V. fig. 2-12-a et b).

B : si dans une séquence, plusieurs paramètres formels possèdent la même valeur actuelle,

et si cela reste vrai au niveau de toutes les séquences du sous-modèle,

alors ces paramètres formels peuvent fusionner (1).

Il nous reste maintenant à savoir quel(s) sous-modèle(s) il faut implémenter, car il paraît évident que l'implantation de deux sous-modèles différents en mémoire MICALL donne des gains différents; en théorie on peut même dire que :

plus un nombre croissant de séquences jouissent de la même représentation en mémoire MICALL,

plus cette représentation est générale (2),

plus le gain individuel est petit.

Le gain au niveau du programme étant le produit du gain individuel et du nombre de répétitions de ce gain, un compromis doit donc être réalisé entre ces deux facteurs; la philosophie adoptée dans notre application est la suivante :

A : Dans chaque modèle du programme, nous détectons tous les sous-modèles et nous les classons par caractéristiques différentes (3); dans chaque classe, nous retenons uniquement le sous-modèle qui contient le plus de séquences.

B : Nous calculons les gains associés à chaque sous-modèle retenu (4).

C : Nous choisissons parmi tous les sous-modèles retenus, ceux qui fournissent le gain maximum; c'est le rôle du programme linéaire (4).

En réalisant le choix globalement au niveau de tous les sous-modèles existant dans le programme, nous sommes assurés de faire le 'meilleur choix'.

Avant de passer au deuxième type de critères de classement, nous donnons une définition; nous appelons schéma de séquences, la description des séquences d'un sous-modèle qui comprend (V. fig. 2-12-b et c):

---

(1) exemple : paramètre formel de la première et de la sixième instruction dans les séquences du sous-modèle 1 (V. fig. 2-12-a et b).

(2) cette généralisation peut être illustrée,par l'exemple, au niveau des paramètres (V. fig. 2-12-c) :
- le premier sous-modèle contient 4 séquences et demande 4 paramètres externes;
- le deuxième sous-modèle contient 3 séquences et demande 2 paramètres externes;
- le troisième sous-modèle contient 3 séquences et demande 3 paramètres externes.

(3) les paramètres internes et les paramètres formels constituent les caractéristiques d'un sous-modèle.

(4) cette partie est réalisée dans la PHASE3 (V. paragraphe 2-4).

SEQUENCES DE DEPART (machine à 1 accumulateur)

| 1 | | | 2 | | | 3 | | | 4 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | LOAD | A | | LOAD | W | | LOAD | L | | LOAD | B |
| | SUB | B | | SUB | B | | SUB | B | | SUB | A |
| | TEST | nul | | TEST | négatif | | TEST | négatif | | TEST | négatif |
| | BC | FIN | | BC | FIN | | BC | FIN | | BC | FIN |
| | STORE | T | | STORE | T | | STORE | T | | STORE | T |
| | LOAD | A | | LOAD | W | | LOAD | L | | LOAD | B |
| | ADD | B | | ADD | B | | ADD | B | | ADD | K |
| | MULT | T | | MULT | T | | MULT | T | | MULT | T |
| FIN : | STORE | A | FIN : | STORE | W | FIN : | STORE | L | FIN : | STORE | B |

Fig. 2-12-a

| Caractéristiques du TYPE | caractéristiques du MODELE | Caractéristiques des SOUS-MODELES (1) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | | 2 | | 3 | | 4 | |
| LOAD | | P1 | | P1 | | P1 | | P1 | |
| SUB | | P2 | | | B | P2 | | | B |
| TEST | | P3 | | P2 | | | négatif | | négatif |
| BC | | | | | | | | | |
| STORE | | | T | | T | | T | | T |
| LOAD | | P1 | | P1 | | P1 | | P1 | |
| ADD | | P4 | | | B | P3 | | | B |
| MULT | | | T | | T | | T | | T |
| STORE | | P1 | | P1 | | P1 | | P1 | |

Fig. 2-12-b

(1) les sous-modèles contiennent respectivement les groupes de séquences (1, 2, 3, 4), (1, 2, 3), (2, 3); la première partie de chaque colonne décrivant les caractéristiques d'un sous-modèle contient les paramètres formels (Pi), la deuxième, les paramètres internes.

| Numéros de SCHEMA | TYPE | MODELE | SOUS-MODELE | PARAMETRES FORMELS | N° de SEQUENCES | PARAMETRES ACTUELS |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | P1, P2, P3, P4 | 1 | A, B, nul, B |
| | | | | | 2 | W, B, négatif, B |
| | | | | | 3 | L, B, négatif, B |
| | | | | | 4 | B, A, négatif, K |
| 2 | 1 | 1 | 2 | P1, P2 | 1 | A, nul |
| | | | | | 2 | W, négatif |
| | | | | | 3 | L, négatif |
| 3 | 1 | 1 | 3 | P1, P2, P3 | 2 | W, B, B |
| | | | | | 3 | L, B, B |
| | | | | | 4 | B, A, K |
| 4 | 1 | 1 | 4 | P1 | 2 | W |
| | | | | | 3 | L |

Fig. 2-12-c

- les caractéristiques du type (liste des codes opérations);
- les caractéristiques du modèle (structure interne des branchements);
- les caractéristiques du sous-modèle (liste des paramètres internes et liste des paramètres formels en correspondance avec la liste des instructions).

2. Si nos seuls critères de classement étaient des critères de regroupement, le volume des informations et le temps passé dans la partie 'choix des sous-modèles à implémenter' (1) deviendraient vite très importants; on peut donc penser qu'il serait préférable d'éliminer (2) au fur et à mesure (3), les classes de séquences (4) qui ont peu de chance d'être implémentées en mémoire MICALL.

   Nous aurions pu baser cette élimination sur le calcul précis des gains réalisés; nous avons préféré, pour des questions de temps d'éxécution (de notre sélecteur), nous baser sur des estimations de ces gains.

   Les critères de rejet que nous avons adoptés peuvent être de trois types :

   A : le critère de répétition basé sur le fait qu'un sous-modèle peu répété produit généralement peu de gain au niveau du programme; ce critère s'énonce :

   si une classe (4) formée par les critères de regroupement contient un nombre de séquences < NBMIN,

   alors si on ne demande pas de considérer les répétitions dynamiques (NOREPDYN = YES),

   alors rejet de la classe;

   sinon si aucune des séquences de la classe n'appartient à une boucle de niveau ≥ BOUNIV,

   alors rejet de la classe;

   sinon accepter la classe;

   sinon accepter la classe.

---

(1) problème de programmation linéaire (V. paragraphe 2-4).

(2) d'où la création de type rejet, modèle rejet, sous-modèle rejet.

(3) en théorie, cela est contraire à une efficacité maximale; en effet la question à se poser n'est pas 'telle séquence vais-je l'implémenter ou la rejeter ?', mais bien 'parmi toutes ces séquences, lesquelles vais-je implémenter et lesquelles vais-je rejeter ?'

(4) ces classes de séquences sont les types, modèles et sous-modèles.

Ce critère de rejet est tout à fait arbitraire et vise uniquement à diminuer la place mémoire occupée par les tables et le temps d'exécution du sélecteur; il se pourrait en effet qu'une séquence non répétée et même à la limite, un petit programme puisse être implémenté en mémoire MICALL s'il reste de la place inoccupée dans celle-ci.
C'est pour cela que nous avons laissé toute la souplesse au niveau de l'utilisateur en lui permettant de paramètrer le rejet (NBMIN, NOREPDYN, BOUNIV) et même de l'annuler en choisissant les 'valeurs neutres' des paramètres (1, YES, 0).

B. les critères de longueur, basés sur le fait que plus le nombre d'instructions regroupées est grand, plus le gain au niveau d'un emploi risque d'être grand (1); il semblerait donc intéressant de ne conserver que la séquence la plus longue d'une famille; malheureusement le gain global dépend aussi du nombre de répétitions, et il est évident que plus une séquence est longue, moins elle est répétée; un compromis doit donc être trouvé.

De nouveau, ce compromis sera réalisé au niveau du choix (V. paragraphe 2-4); lors de l'extension des séquences, un premier critère, appelé critère de la longueur maximum ne conservera parmi toutes les séquences d'une famille que celle qui a la plus grande longueur pour un nombre de répétitions fixées (2).

Un deuxième critère, appelé critère de la longueur minimum, fixera la longueur minimum d'une séquence, à LGMIN; ce critère est tout aussi arbitraire que le critère de répétition puisque, à la limite, il peut être intéressant d'implémenter en mémoire MICALL une instruction qui reviendrait toujours avec le même paramètre.

C. le critère de standardisation d'interfaces; une séquence telle que nous l'avons décrite peut posséder plusieurs entrées et plusieurs sorties (3).

Lorsque l'on implémente de telles séquences en mémoire MICALL, les interfaces de sorties posent peu de problèmes, car ils se résument généralement à un positionnement de codes-conditions et à une mise à jour du compteur de programme.

---

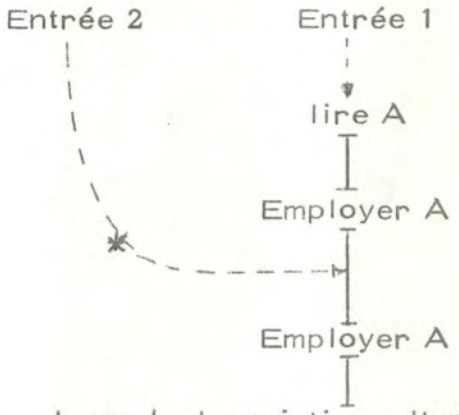
(1) par exemple : le rapport entre le nombre de paramètres de la séquence d'instructions initiale et le nombre de paramètres formels du schéma sera beaucoup plus grand.

(2) ces répétitions sont des répétitions de séquences d'un même type mais appartenant à des familles différentes.

(3) c'est-à-dire qu'elle peut posséder plusieurs instructions référencées de l'extérieur et plusieurs branchements vers l'extérieur.

Par contre, les interfaces d'entrées sont beaucoup plus compliqués; si l'on permet plusieurs entrées, des problèmes délicats peuvent se poser lors de l'implémentation du programme en mémoire (MICALL, notamment en ce qui concerne la lecture des paramètres; souvent, une nouvelle entrée demande des traitements supplémentaires : lectures mémoire calculs d'adresses ... (V. fig. 2-13), ce qui entraîne une perte d'efficacité.

Pour cette raison, nous avons décidé, dans notre application de rejeter systématiquement toutes les séquences possédant plusieurs entrées



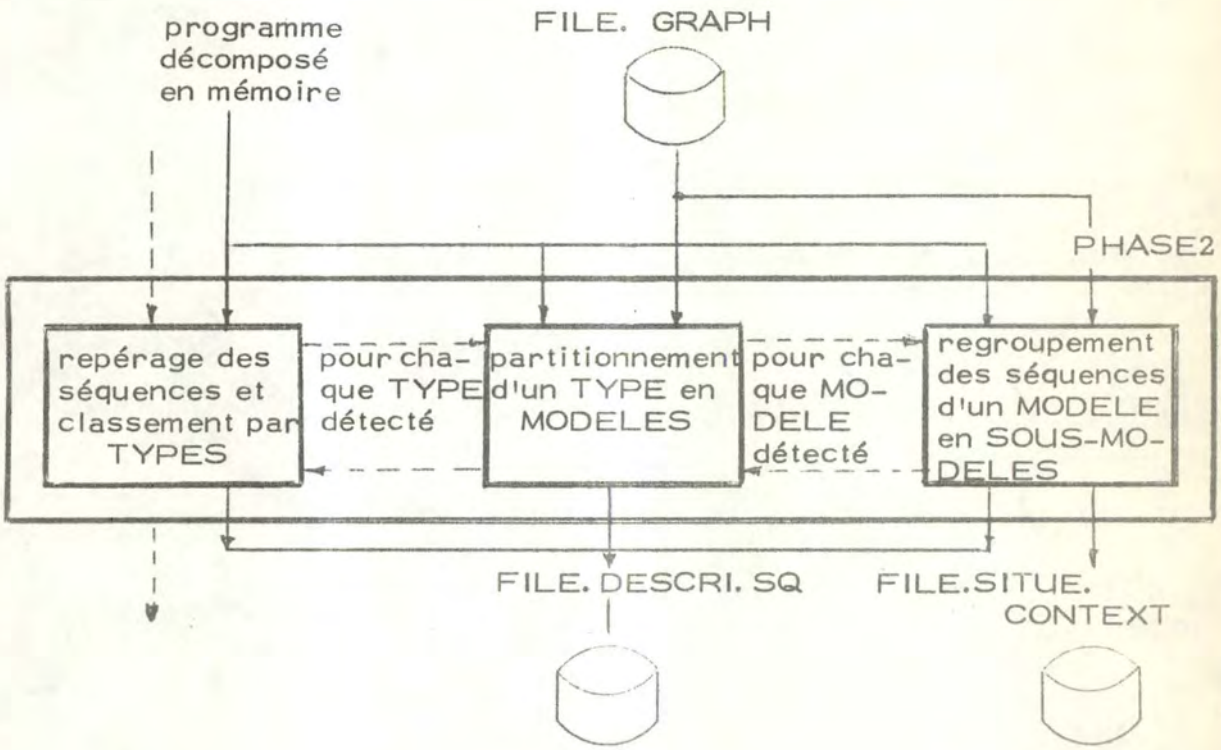

Légende ⊢——⊣ description d'une partie de traitement en mémoire MICALL

* nécessité d'introduire une lecture de A.

Fig. 2-13

## 2-3-3. ALGORITHMES DE REPERAGE ET CLASSEMENT.

Dans les paragraphes précédents, nous avons tout d'abord décrit le principe du repérage des séquences (2-3-1) puis exposé les différents critères employés pour classer les séquences repérées (2-3-2) ; nous nous intéresserons ici à l'organisation du 'repérage et classement'.




Légende ——→ utilisation ou production de fichiers

- - -→ séquence de l'exécution.

Fig. 2-14 (1)

Nous utiliserons trois grandes parties (V. fig. 2-14), chacune employant un critère de regroupement et un ou plusieurs critères de rejet :

---

(1) l'utilisation des fichiers doit être mise en rapport avec le rôle joué par les différentes parties de la PHASE 2.

1. La première est la plus compliquée parce qu'elle combine à la fois : le repérage des séquences par familles, le classement des séquences par types et le rejet de certains types à partir du critère de répétition, du critère de la longueur maximum et du critère de la longueur minimum.

   Pour mieux comprendre le principe de cette partie, nous commencerons par suivre pas à pas, sur un exemple, l'algorithme employé; la liste des codes opérations du programme (1) et la progression de l'algorithme sont décrits respectivement dans les figures 2-15 et 2-16.
   Dans une première étape (V. fig. 2-16) nous repérons toutes les séquences de longueur 1 et nous les classons par types.
   Dans l'étape 2-1, nous étendons de 1 chaque séquence appartenant au premier type de la première étape (H) et nous classons les nouvelles séquences par types ( (H, G) et (H, J) ).
   De même, dans les étapes 2-i, nous étendons de 1 chaque séquence appartenant à un type de la première étape, à condition qu'il vérifie le critère de répétition (2); il est en effet évident que si un type de la première étape ne vérifie pas le critère de répétition, les types qui pourraient en découler par extension ne vérifieraient pas non plus ce critère, nous abandonnons donc (Ab) l'extension des séquences de ce type (3).
   Lorsque l'extension des séquences de la première étape prend fin, on commence à étendre les séquences de la deuxième étape et ainsi de suite jusqu'au moment où tous les types d'une étape doivent être abandonnés (4).
   Lorsque l'on abandonne l'extension des séquences d'un type (par exemple, le type (H, G, J) de l'étape 3-1), deux choses sont à remarquer :

   - le type que l'on abandonne (H, G, J) ne vérifie pas le critère de répétition (car il ne contient qu'une seule séquence); ce type doit donc être rejeté;

   - le type dont provient le type abandonné (H, G) vérifie le critère de répétition (car il contient 3 séquences).

   Le type dont provient le type abandonné (H, G) est donc un des types qui n'est pas rejeté par le critère de la longueur maximum et l'on peut même dire (V. déf. p. 2-25) que tous les types non rejetés par ce critère peuvent être détectés de cette manière.
   Le critère de la longueur minimum provoque un dernier rejet parmi les types restants(5) .
   Finalement, les types non rejetés sont décrits dans la fig. 2-16 par la liste des codes opérations et les adresses d'occurrence des séquences qu'ils contiennent.

---

(1) les codes opérations sont notés : A, B. . . , K L; le symbole ## représente la fin de liste.
(2) c'est-à-dire, dans le cas de l'exemple; 'qu'il contienne au moins 2 séquences'.
(3) l'abandon correspond à un premier rejet.
(4) dans notre exemple, cela se passe à l'étape 4.
(5) c'est ainsi que toutes les séquences abandonnées à l'étape 2 donnent des types rejetés.

| Adresses | 1 | | | | 5 | | | | | 10 | | | | | 15 | | | | | 20 | | | 23 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Liste des codes opérations | H | G | A | B | C | D | E | H | G | A | F | C | D | J | A | B | C | H | G | J | H | J | # |
| Séquences retenues | | | | | | | | | | | | | | | | | | | | | | | |

Fig. 2-15

Programme et liste des séquences retenues

| Etape | Caractéristiques du TYPE | Adresses des séquences appartenant à ce TYPE | | TYPES non rejetés |
|---|---|---|---|---|
| 1 | H | 1, 8, 18, 21 | | |
| | G | 2, 9, 19 | | |
| | A | 3, 10, 15 | | |
| | B | 4, 16 | | |
| | C | 5, 12, 17 | | |
| | D | 6, 13 | | |
| | E | 7 | Ab | |
| | F | 11 | Ab | |
| | J | 14, 20, 22 | | |
| | # | 23 | Ab | |
| 2-1 | H, G | 1, 8, 18 | | |
| | H, J | 21 | Ab | |
| 2-2 | G, A | 2, 9 | | |
| | G, J | 19 | Ab | |
| 2-3 | A, B | 3, 15 | | |
| | A, F | 10 | Ab | |
| 2-4 | B, C | 4, 16 | | |
| 2-5 | C, D | 5, 12 | | |
| | C, H | 17 | Ab | |
| 2-6 | D, E | 6 | Ab | |
| | D, J | 13 | Ab | |
| 2-7 | J, A | 14 | Ab | |
| | J, H | 20 | Ab | |
| | J, # | 22 | Ab | |
| 3-1 | H, G, A | 1, 8 | | |
| | H, G, J | 18 | Ab | TYPE1 H, G (1, 8, 18*) |
| 3-2 | G, A, B | 2 | Ab | TYPE2 G, A (1*, 9*) |
| | G, A, F | 9 | Ab | |
| 3-3 | A, B, C | 3, 15 | | |
| 3-4 | B, C, D | 4 | Ab | TYPE3 B, C(4*, 16*) |
| | B, C, H | 16 | Ab | |
| 3-5 | C, D, E | 5 | Ab | TYPE4 C, D(5*, 12*) |
| | C, D, J | 12 | Ab | |
| 4-1 | H, G, A, B | 1 | Ab | TYPE5 H, G, A(1*, 8*) |
| | H, G, A, F | 8 | Ab | |
| 4-2 | A, B, C, D | 3 | Ab | TYPE6 A, B, C(3*, 15*) |
| | A, B, C, H | 15 | Ab | |

Légende * repère les adresses des séquences ayant provoqué l'abandon.
Ab signifie 'abandon de l'extension des séquences d'un type pour qu'il ne vérifie plus le critère de répétition'.

Fig. 2-16

Application de l'algorithme à l'exemple de la fig. 2-15 pour les valeurs (2, 2, YES) des paramètres (LGMIN, NBMIN, NOREPDYN)

D'une façon plus formelle, nous décrirons la première étape de la manière suivante :

première partie.

Initialisation :

A : - repérer dans la suite des codes opérations toutes les séquences de longueur 1 et les classer par types;
- LGCUR = 0;
- aller en B.

Abandon et rejet :

B : pour chaque type que l'on vient de repérer,
si le type ne vérifie pas le critère de répétition,
alors - abandonner l'extension des séquences de ce type;
- rejeter ce type et tous ceux qui suivent ;
- si LGCUR < LGMIN (critère de la longueur minimum),
alors passer au type suivant (1);
sinon - accepter un nouveau type (2);
- appeler la deuxième partie pour créer les modèles;
- passer au type suivant (1);
sinon rejeter ce type (critère de la longueur maximum);
- passer au type suivant (1);
à la fin aller en C.

Détection de la fin de l'algorithme :

C : si l'étape B a abandonné tous les types,
alors fin de l'algorithme;
sinon aller en D.

Création d'une nouvelle étape :

D : - LGCUR = LGCUR + 1;
- pour chaque type non abandonné par l'étape B, étendre toutes les séquences de 1 et les regrouper dans de nouveaux types;
à la fin aller en B.

2. La deuxième partie se charge de partitionner en modèles chaque type qu'elle reçoit; outre le critère de regroupement en modèles, cette partie utilise deux critères de rejet ; le critère de répétition et le critère de standardisation d'interfaces; l'algorithme qui représente cette partie peut être décrit de la manière suivante :

---

(1) passer au type suivant signifie continuer la boucle qui se trouve en B.
(2) le type accepté est celui dont découle directement par extension, le type abandonné.

Deuxième partie :

pour chaque séquence contenue dans le type,

- détecter sa structure interne de branchements;
- si la séquence ne vérifie pas le critère de standardisation d'interfaces,

  alors - rejeter cette séquence;

  - passer à la séquence suivante;

  sinon si la structure interne de branchements correspond à celle d'un modèle (1) déjà créé,

  alors - rajouter la séquence à ce modèle;

  - passer à la séquence suivante;

  sinon - créer un nouveau modèle;

  - passer à la séquence suivante;

à la fin pour chaque modèle,

si le modèle ne vérifie pas le critère de répétition,

alors rejeter le modèle;

sinon - appeler la troisième partie pour créer des sous-modèles,

- passer au modèle suivant;

à la fin retourner à la première partie.

3. La troisième partie se charge de la création des sous-modèles; elle utilise le critère de regroupement en sous-modèles et le critère de rejet concernant les répétitions; l'algorithme est le suivant :

Troisième partie :

- créer tous les sous-modèles du modèle, en tenant compte des deux règles de la page 2-20 .
- pour chaque sous-modèle,

  si le sous-modèle ne vérifie pas le critère de répétition,

  alors - rejeter le sous-modèle;

  - passer au sous-modèle suivant;

  sinon passer au sous-modèle suivant ;

  à la fin retour à la deuxième partie.

---

(1) il s'agit bien sûr, d'un modèle appartenant au même type.

## 2-4. PHASE 3 : CHOIX DES SEQUENCES A IMPLEMENTER.

2-4-0. La situation au début de cette phase est la suivante :

- pour chaque adresse du programme, on connaît la liste des schémas qui représentent une séquence commençant à cette adresse (fichier FILE. SITUE. CONTEXT);
- on connaît les caractéristiques de chaque schéma (FILE. DESCRI. SQ).

On voudrait connaître les schémas à implémenter pour avoir le gain maximum au niveau du programme en tenant compte que la place en mémoire MICALL est limitée.

Le problème posé ressemble donc au problème de la répartition d'un budget entre différents projets (R 34) et peut s'énoncer :

'On possède un certain budget à investir (place disponible en mémoire MICALL : $C_{micall}$);
un certain nombre de projets sont proposés (projet d'implanter en mémoire MICALL le schéma j);
chaque projet demande un investissement de départ (place occupée par l'implantation en mémoire MICALL : Cj),
pour chaque projet, on a estimé le profit que l'on pourrait en retirer (gain mémoire utilisateur et gain en temps d'exécution réalisés au niveau du programme : Gj);
on se demande quels projets on va sélectionner pour avoir le profit maximum, en tenant compte que le projet ne peut être sélectionné au maximum qu'une seule fois. '

Il s'agit donc d'un problème de programmation mathématique en nombres entiers et en variables 0 ou 1; formellement, il s'énonce comme suit (1) :

$$\max.\ G = \sum_j \delta_j\, G_j \qquad \delta_j\ ?$$

(j parcourt l'ensemble des schémas du programme)

sous les contraintes :

$$\sum_j \delta_j\, C_j \leqslant C_{micall}$$

$$\delta_j = 0 \text{ ou } 1.$$

Mais en réalité, d'autres contraintes doivent être ajoutées ; il s'agit des contraintes d'exclusion entre les schémas; nous les décrirons dans les paragraphes suivants ; disons déjà qu'elles ont la forme : $\sum \delta_i \leqslant 1$ (où i parcourt un sous-ensemble de l'ensemble des schémas du programme.)

---

(1) $\delta_j = 0$ si le schéma n'est pas implémenté et = 1 dans le cas contraire;

La fig. 2-17 décrit les principales parties de la configuration adoptée pour la PHASE3; c'est-à-dire :

- la création des contraintes;
- la création de la fonction économique;
- le problème de programmation mathématique;
- le programme de modification des fichiers.

Dans les paragraphes suivants (2-4-1, 2-4-2), nous ne décrirons que la logique des deux premières parties, la troisième pouvant être résolue par un programme standard existant et la quatrième se résumant à une simple suppression d'enregistrements dans des fichiers.

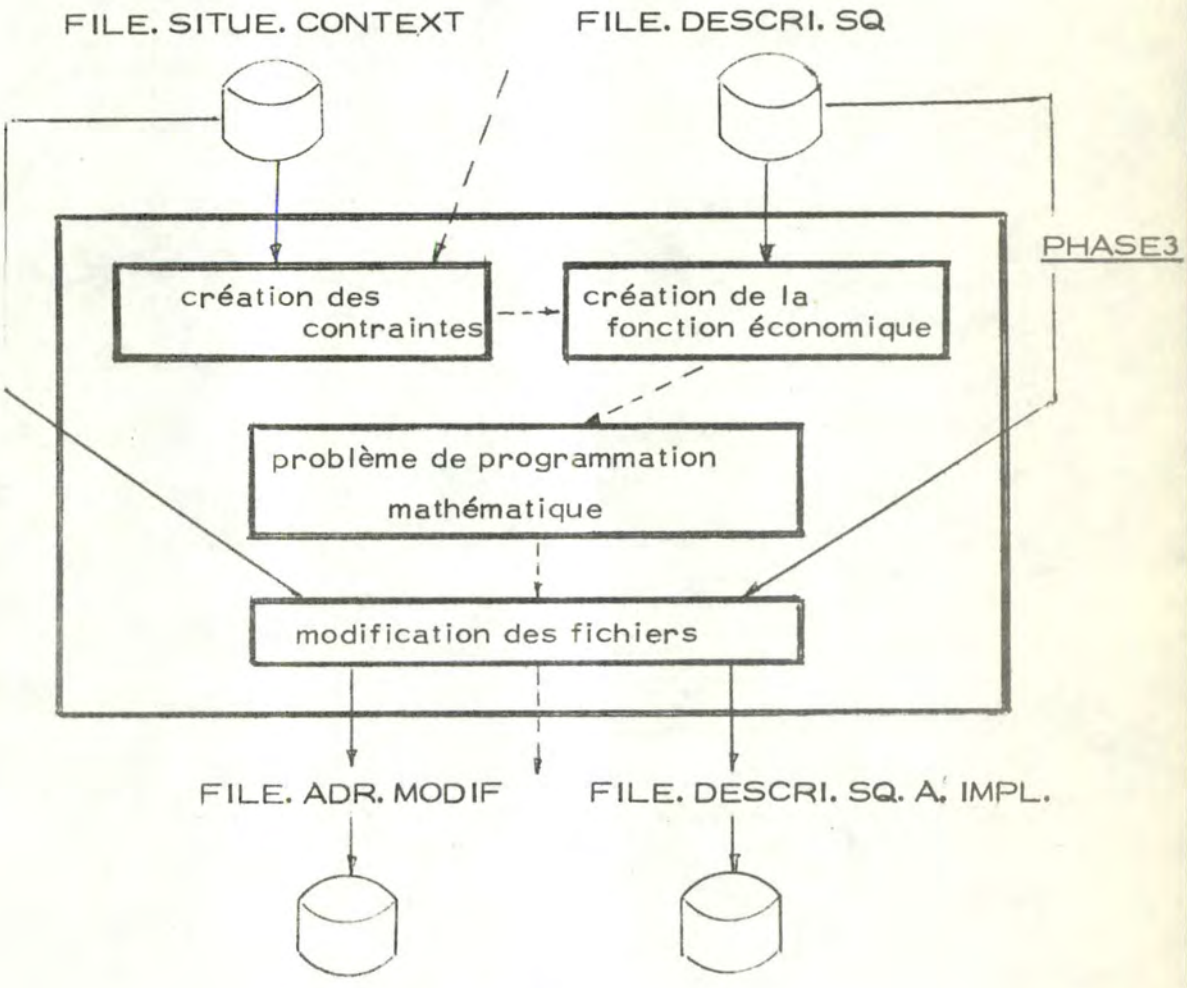



Fig. 2-17

⟶ utilisation des fichiers et création.

⟶ séquence de l'exécution.

## 2-4-1. CREATION DE LA FONCTION ECONOMIQUE.

La description de la fonction économique : $\leq \delta j$ Gj , demande quelques précisions au sujet du calcul du gain Gj (1) associé à l'implantation d'un schéma en mémoire MICALL.

Celui-ci peut être décomposé en deux parties :

- le gain en place 'mémoire utilisateur' occupée : GMMj (1) qui s'exprime en nombre de mots;
- le gain en temps d'exécution : GTj (1) qui s'exprime en $\mu$s.

Le gain Gj peut alors être décrit par :

$$Gj = K1 \left[1/mot\right] * GMMj \left[mot\right] + K2 \left[1/\mu s\right] * GTj \left[\mu s\right] ,$$

où K1 et K2 sont deux constantes qui ont pour rôle :

- d'ajuster les échelles '$\mu$s' et 'mot' et de permettre ainsi d'exprimer le gain en absolu (pas d'unité);
- de permettre d'équilibrer le compromis temps/place en accentuant plus ou moins l'influence de chacun des deux termes.

Le calcul de ces constantes se révèle assez compliqué parce qu'il est influencé par le type de calcul des nombres GMMj et GTj; aussi avons-nous opté pour une recherche empirique de ces constantes.

Les gains GMMj et GTj sont réalisés au niveau du programme origine, par l'implémentation du schéma j en mémoire MICALL suivie du remplacement des séquences d'instructions du programme par un appel au programme généré en mémoire MICALL.

Ces gains proviennent en fait des gains unitaires GMM1j et GT1j calculés pour le remplacement d'une séquence du programme origine par l'appel correspondant :

- Le calcul du gain mémoire GMM1j est évident puisque l'on connaît la séquence d'instructions du programme origine et le nombre de paramètres de l'appel (2);
- Quant au calcul des gains en temps GT1j, il est facilité par la méthode de génération que nous avons choisie (3). Une table du système met en correspondance chaque instruction machine avec tous les modules dont elle est fabriquée (V. ch. 1), leurs temps respectifs d'exécution (4) et les conditions dans lesquelles ils deviennent inutiles;

---

(1) ce gain est réalisé au niveau du programme.

(2) ceux-ci sont décrits dans la partie 'caractéristiques du sous-modèle'.

(3) génération en mémoire MICALL d'appels de module préfabriqués se trouvant en mémoire MIDEF (V. ch. 1).

(4) ces temps peuvent dépendre des données; il s'agira alors de temps moyens.

elle nous permet de prévoir grossièrement la génération d'un schéma, de connaître les modules qui disparaissent (1) et d'estimer ainsi le temps gagné au niveau d'un appel (2). Dans cette estimation, nous devons tenir compte des boucles internes qui augmentent le gain en temps lorsqu'un module qu'elles contiennent est supprimé (3); malheureusement cette augmentation nous est généralement inconnue car elle dépend des données; la seule chose que nous pouvons faire, c'est pondérer le gain par une fonction dépendant du niveau d'imbrication de la boucle; mais il faut être prudent dans le choix de cette pondération parce qu'elle peut mener à des erreurs graves. De plus, le gain unitaire en temps GT1j doit lui-même être pondéré lorsque la séquence remplacée dans le programme appartient à une boucle de ce programme; cette pondération est tout aussi arbitraire et donc dangereuse que celle décrite ci-dessus, mais elle est nécessaire.

Pour calculer les gains réalisés au niveau du programme origine GMMj et GTj, il suffit de sommer les gains unitaires GMM1j et GT1j sur le nombre de remplacements de séquences d'instructions (par un appel à la mémoire MICALL) effectivement réalisés dans le programme origine; il est nécessaire de bien voir la différence entre :

- le nombre de séquences représentées par le schéma (4) et
- le nombre de remplacements effectivement réalisés lors d'une implémentation;

ce dernier tient compte en effet, des différents schémas implémentés simultanément et en particulier des positions relatives des séquences qu'ils contiennent; en effet, si deux séquences non disjointes (5) dans le programme ont deux représentations différentes possibles (deux schémas (V. fig. 2-18)), une seule séquence et une seule représentation seront effectivement choisies lors du remplacement et le nombre de remplacements effectifs d'un des deux schémas sera diminué de 1 (6).

---

(1) la plupart du temps, il s'agit de modules de lecture de paramètres et de calculs d'adresses; leur disparition est liée au gain mémoire.

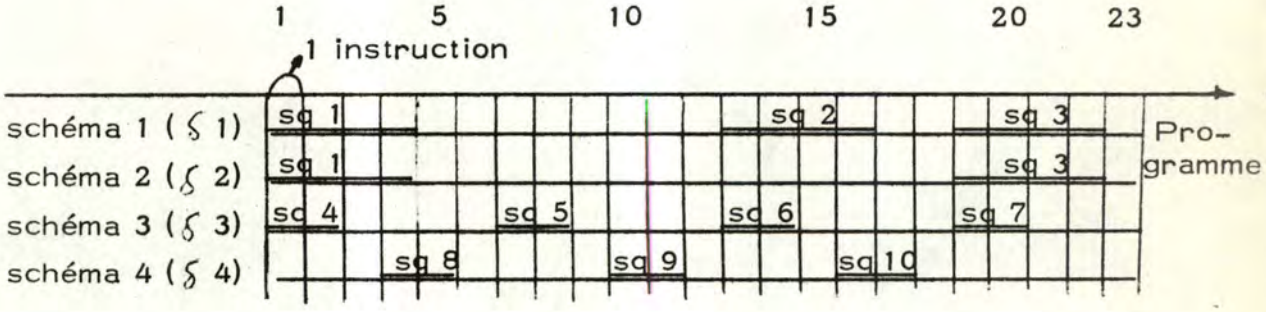
(2) il suffit de faire la somme des temps de chaque module supprimé en tenant compte, éventuellement, des parties se déroulant en simultanéité.

(3) il faut en plus remarquer que l'implantation des schémas ayant une longueur statique (nombre d'instructions décrites) minimum et une longueur dynamique (nombre d'instructions exécutées) maximum améliore en principe l'utilisation de la mémoire MICALL.

(4) ce nombre est, en fait, le nombre de séquences appartenant au sous-modèle correspondant au schéma; il peut être calculé en PHASE2.

(5) à la limite on peut considérer deux autres cas :
- une séquence appartenant à deux schémas différents (ce point sera développé dans la suite) (séquence 1 de la fig. 2-18)
- deux séquences non disjointes appartenant au même schéma; ce cas, bien que rare en pratique mérite tout de même d'être signalé (par exemple, la suite d'instructions L, ST, L, ST, L contient deux séquences L, ST, L non disjointes, mais appartenant au même schéma).

(6) par exemple (V. fig. 2-18), la séquence 1 et la séquence 8 ne sont pas disjointes et sont représentées respectivement par les schémas (1, 2) et 4; si l'on choisit de représenter la séquence 1 par le schéma 1, le nombre de remplacements effectifs des schémas 2 et 4 sera diminué de 1.




- $\delta$ i = variable du problème de programmation mathématique ;
- sqi = séquence ;
- l'axe orienté représente la suite d'instructions du programme ;
- chaque ligne représente les séquences appartenant à un même schéma.

Fig. 2-18

Le nombre de remplacements effectifs correspondant à un schéma n'est donc connu qu'après le choix, ce qui pose de sérieux problèmes; plusieurs solutions sont possibles pour les résoudre :

- La première consiste à exclure entre eux les schémas représentant des séquences non disjointes ou des séquences communes; on rajoute alors des contraintes d'exclusion (1) et l'on prend pour nombre de remplacements effectifs, le nombre de séquences représentées par le schéma; un simple exemple (2) va nous prouver l'inefficacité d'une telle solution; supposons que nous ayons les trois schémas dont les caractéristiques sont reprises à la fig. 2-19; en employant la solution décrite ci-dessus, on génère une contrainte $\delta 1 + \delta 2 < 1$ et on implémente en mémoire

| | portion de la mémoire MICALL occupée | nombre de séquences représentées | gain unitaire | Interférence |
|---|---|---|---|---|
| schéma 1 ( $\delta$ 1) | 2/3 | 9 sq. | 10 | |
| schéma 2 ( $\delta$ 2) | 1/3 | 4 sq. | 20 | avec le schéma 1 (3 sq) |
| schéma 3 ( $\delta$ 3) | 1/3 | 3 sq. | 2 | |

Fig. 2-19

---

(1) dans l'exemple nous aurions les contraintes (V. fig. 2-18)
$\delta 1 + \delta 2 + \delta 3 \leqslant 1$ et $\delta 1 + \delta 2 + \delta 4 \leqslant 1$.

(2) cet exemple sera bien sûr fort simplifié, mais les principes restent les mêmes dans un cas pratique.

MICALL le schéma 1 qui donne un gain de 90 et le schéma 3 qui donne un gain de 6; le gain total réalisé est donc de 96; par contre, si l'on décidait d'implémenter les 4 séquences du schéma 2 (gain = 80) et les 6 séquences restantes du schéma 1 (gain 60), le gain total serait de 140.

- Cette deuxième méthode peut s'énoncer comme suit :

  ' si deux schémas (1) interfèrent (2) entre eux, et si l'on désire les implémenter simultanément, les séquences d'instructions disjointes sont remplacées par l'appel correspondant à leur schéma respectif et les séquences non disjointes, par l'appel correspondant au schéma offrant le plus grand gain unitaire'; en pratique, cependant, cette méthode ne peut nous satisfaire car elle rend la fonction économique non linéaire (3).

- Une troisième solution consiste à linéariser la fonction économique précédente; pour cela, chaque fois qu'il y a interférence entre plusieurs schémas, nous considérons toutes les combinaisons possibles de ces schémas comme étant implémentables (4) et nous rajoutons une contrainte excluant toutes les interférences parmi les combinaisons et les schémas de départ (5). Une combinaison est donc un groupe fixé de schémas qui interfèrent entre eux; l'interférence étant prévue avant le choix (6), nous pouvons calculer (7) le nombre de remplacements qui a effectivement lieu dans chaque schéma de la combinaison, si on décide (dans le programme de choix) d'implémenter celle-ci; ainsi, les gains Gj sont connus en absolu et la fonction économique est donc linéaire.
  Le seul désavantage de cette méthode est l'augmentation du nombre de variables $\delta$ i et du nombre de contraintes.

---

(1) la généralisation à plus de deux schémas est immédiate.

(2) on dit que deux schémas interfèrent entre eux lorsqu'ils possèdent au moins une séquence commune ou deux séquences (chacun une) non disjointes.

(3) si l'on reprend l'exemple de la fig. 2-19, les coefficients de la fonction économique (Gj) dépendent en effet des variables $\delta$ i provenant des 'nombres de remplacements effectifs' (9-3 $\delta$1, 4, 2).

(4) à chaque combinaison est donc associée une nouvelle variable $\delta$ i, lors du choix on pourra donc décider d'implémenter soit un schéma, soit une combinaison de schémas (groupe fixé de schéma).

(5) cette contrainte d'exclusion a le même rôle que celle décrite dans la première solution, la troisième solution est donc inspirée des deux précédentes.

(6) puisque une combinaison est un groupe fixé de schémas.

(7) ce calcul est inspiré de la deuxième méthode.

Dans le paragraphe suivant, nous allons détailler la génération des combinaisons et des contraintes d'exclusion.

## 2-4-2. GENERATION DES CONTRAINTES.

Comme nous l'avons déjà dit, nous considérons deux types de contraintes :

- la contrainte de limitation de capacité de la mémoire MICALL ($\sum_j C_j \leq C_{micall}$) ; un coefficient Cj de cette contrainte représente la place qu'occuperait en mémoire MICALL, le schéma ou la combinaison correspondante; il peut être estimé facilement puisque l'on connaît les modules intervenant dans la génération d'un schéma (V. paragraphe 2-4-1);

- les contraintes d'exclusion qui proviennent des interférences entre les schémas (V. paragraphe 2-4-1)(1);la génération de ces contraintes se fait parallèlement à la génération des combinaisons (2); elle est réalisée par un parcours séquentiel du fichier FILESITUE. CONTEXT en tenant compte à chaque pas (3) des interférences; celles-ci peuvent être détectées grâce à la connaissance à chaque pas :

  - des séquences commençant à cette adresse et du schéma correspondant;

  - de la liste des schémas en cours.

  L'algorithme décrit ci-après peut être suivi (sur la fig. 2-20) pour l'exemple décrit à la fig. 2-18 :

---

(1) deux types d'interférences particulières méritent d'être signalés :
- une même séquence peut être représentée par plusieurs schémas caractérisés par les mêmes types et modèles, mais par des sous-modèles différents; ce type d'interférences entre schémas provient du fait que le compromis nombre de paramètres formels (c'est-à-dire place mémoire occupée par appel)/ nombre de répétitions (V. paragraphe 2-3) n'a pas encore été résolu;
- chaque séquence d'un modèle peut contenir une séquence de la même famille appartenant à un autre modèle (unique); cela est dû au fait que le compromis longueur des séquences/nombre de répétitions n'a pas encore été résolu (V. paragraphe 2-3).

(2) et donc des variables $\delta_i$ associées.

(3) une entrée de ce fichier correspond à une adresse du programme et contient entre autre pour chaque séquence commençant à cette adresse dans le programme origine, la longueur de la séquence et le ou les numéros des schémas qui la représentent.

pour chaque entrée du fichier FILE. SITUE. CONTEXT (1)

si l'entrée du fichier ne contient aucun numéro de schémas (2),

alors passer à l'entrée suivante;

sinon - générer toutes les combinaisons possibles des schémas appartenant à l'entrée et à la liste à condition que ces combinaisons n'aient pas déjà été générées (3),

- si l'on a généré au moins une nouvelle combinaison,

alors générer une contrainte d'exclusion entre toutes les nouvelles combinaisons créées et les schémas intervenant dans ces combinaisons; aller en A;

sinon aller en A;

A : - mettre les numéros des schémas de l'entrée, dans la liste;

- supprimer de la liste les numéros de schémas dont les séquences correspondantes se terminent (4);

- passer à l'entrée suivante;

à la fin fin de l'algorithme.

---

(1) équivalent à : pour chaque adresse du programme.

(2) c'est-à-dire qu'aucune séquence retenue ne commence à cette adresse.

(3) les combinaisons déjà générées sont notamment celles utilisant uniquement des éléments de la liste.

(4) on peut savoir quand une séquence se termine puisque l'on connaît l'endroit où elle commence et sa longueur.

| Adresse dans le programme | schémas d'entrée (1) | liste des schémas en cours | Combinaisons et contraintes d'exclusions créées |
|---|---|---|---|
| 1 | 1, 2, 3 | - | C1(1, 2); C2(1, 3); C3(2, 3); C4(1, 2, 3); EX(1, 2, 3, C1, C2, C3, C4); |
| 2 | - | 1, 2, 3 | |
| 3 | - | 1, 2 | |
| 4 | 4 | 1, 2 | ~~C1(1, 2);~~ C5(1, 4); C6(2, 4); EX(1, 2, 4, C1, C5, C6); |
| 5 | - | 4 | |
| 6 | - | - | |
| 7 | 3 | - | |
| 8 | - | 3 | |
| 9 | - | - | |
| 10 | 4 | - | |
| 11 | - | 4 | |
| 12 | - | - | |
| 13 | 1, 3 | - | ~~C2(1, 3)~~ |
| 14 | - | 1, 3 | |
| 15 | - | 1 | |
| 16 | 4 | 1 | ~~C5(1, 4)~~ |
| 17 | - | 4 | |
| 18 | - | - | |
| 19 | 1, 2, 3 | - | ~~C1(1, 2); C2(1, 3), C3(2, 3); C4(1, 2, 3)~~ |
| 20 | - | 1, 2, 3 | |
| 21 | - | 1, 2 | |
| 22 | - | 1, 2 | |
| 23 | - | - | |

C1(1, 2) désigne la combinaison 1 contenant les schémas 1 et 2.

EX(1, C1, 3) identifie une contrainte d'exclusion entre le schéma 1, la combinaison C1 et le schéma 3.

Fig. 2-20

---

(1) V. fichier FILE. SITUE. CONTEXT.

# CHAPITRE 3

## IMPLANTATION PHYSIQUE DU SELECTEUR

3-0. Nous venons de décrire les grands problèmes logiques soulevés par le sélecteur; nous nous proposons dans les paragraphes suivants d'examiner son implantation physique.
Elle peut être vue à deux niveaux que nous décrirons parallèlement :

- structure physique du programme;
- organisation des tables en mémoire.

A un niveau assez général, nous pouvons déjà dire que :

- la structure logique du programme (V. fig. 2-1) se conserve parfaitement lors de l'implémentation;
- la mémoire est organisée en plusieurs 'blocs' (V. fig. 3-1), chaque phase et ses données propres sont contenues dans des zones de 4KB maximum, dont l'adresse de base est donnée par le registre BASPROG; une zone commune de travail, ZOON, contiendra les tables, elle peut atteindre 64KB et son adresse de base est donnée par le registre RBASE.

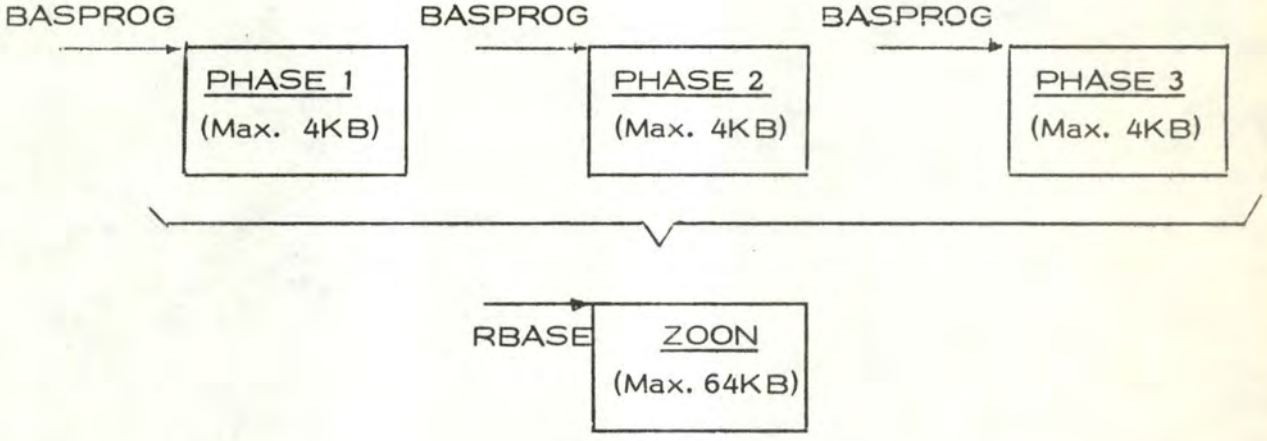



Fig. 3-1

Organisation générale de la mémoire

Avant de poursuivre tous développements, nous allons citer quelques options d'implantation que nous avons prises :

- nous avons choisi l'ordinateur Siemens 4004 pour implémenter le sélecteur (1) ; tenant compte de cette option, une configuration s'est imposée (V. fig. 3-2) (2);
- le sélecteur est écrit en langage machine, ce qui se justifie d'abord pour réduire le volume des tables, ensuite pour réduire les temps d'exécution;
- les parties en rapport avec les répétitions dynamiques sont prévues mais non implémentées (3) ;
- le problème de programmation linéaire est résolu par un programme standard existant;
- la description de l'implémentation physique de la PHASE3 n'est pas reprise ici, mais elle paraît assez évidente.

---

(1) l'ordinateur VARIAN n'étant pas encore opérationnel.

(2) dans cette configuration, les intermédiaires entre les deux machines sont des fichiers cartes, ce qui enlève une certaine souplesse d'emploi; il faut cependant se rappeler que l'optimisation ne doit en théorie, se passer qu'une seule fois dans la 'Vie' d'un programme et seulement pour des programmes fréquemment exploités.

(3) le test de NOREPDYN permettra de diriger l'exécution selon les besoins; nous n'avons pas implémenté cette partie pour diverses raisons dont la principale est que le gain attendu par les répétitions dynamiques paraît moins évident que celui des répétitions statiques.

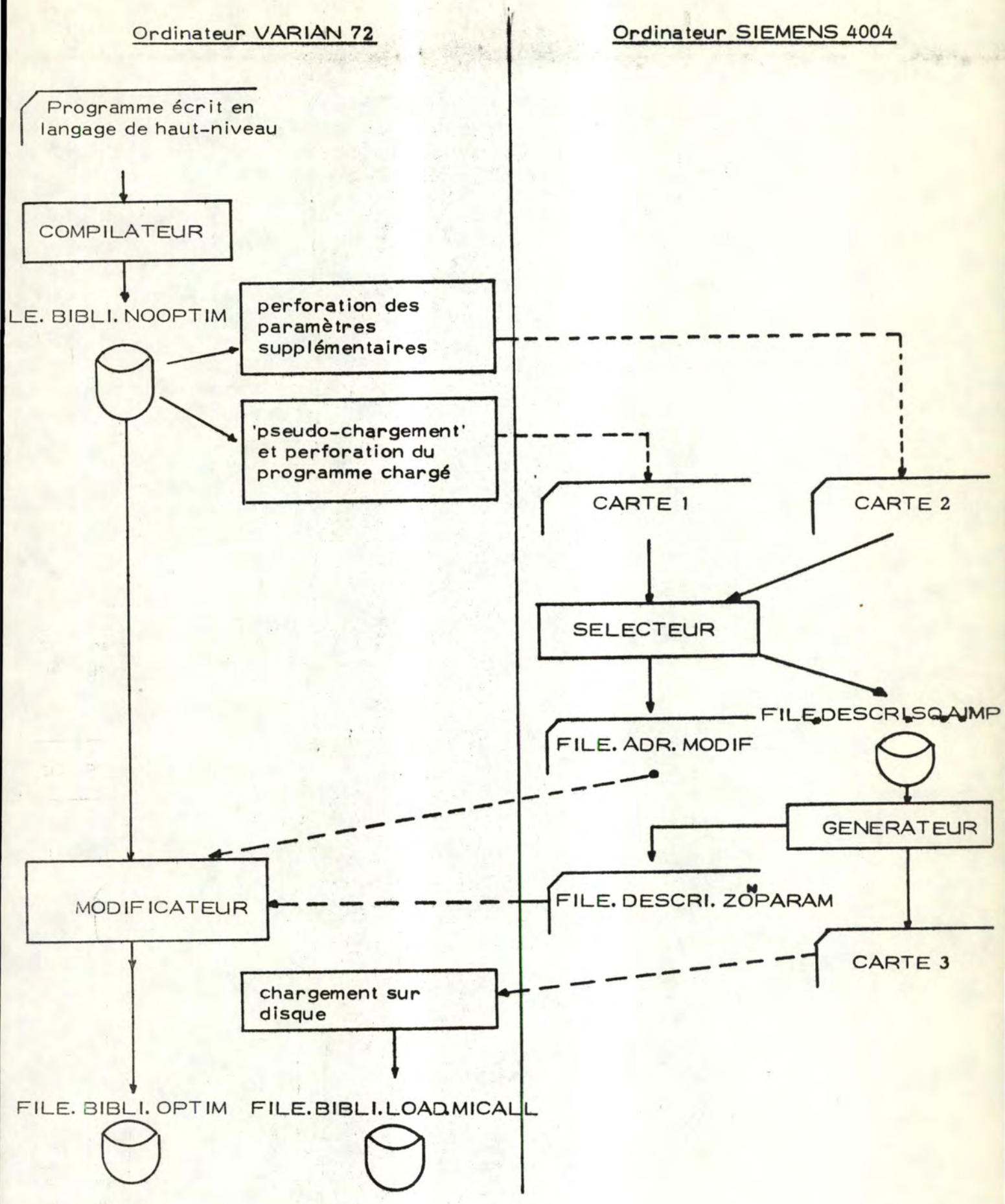



Fig. 3-2

Description d'une configuration possible

## 3-1. IMPLEMENTATION PHYSIQUE DE LA PHASE 1 : 'ANALYSE DE LA STRUCTURE DU PROGRAMME'.

3-1-0. La structure physique (V. fig. 3-3a) de la PHASE 1 se déduit directement de sa structure logique (V. fig. 2 - 2); nous remarquerons cependant :

- que la création du graphe se fait en deux parties dont une partie d'initialisation;
- qu'un test de NOREPDYN permet de ne pas exécuter la partie 'détection des circuits élémentaires' dans le cas où l'on ne désire pas considérer les répétitions dynamiques;
- que deux parties supplémentaires PPHASE1D et PPHASE1E créent les fichiers d'interface entre la PHASE1 et la PHASE2 et entre la PHASE1 et l'utilisateur.

Dans la fig. 3-3b., nous décrivons l'évolution de la zone commune de mémoire durant les différentes parties de la PHASE1; cette évolution est directement en rapport avec le traitement opéré durant ces parties de la PHASE1.

Dans les paragraphes suivants nous décrirons successivement :

- les fichiers d'entrées CARTE1, CARTE2 (3-1-1);
- le fichier FILE. MOD. CHARG et la nouvelle structure d'adressage qu'il entraîne (3-1-2);
- la représentation du graphe en mémoire et la description des modules associés à sa gestion (3-1-3);
- les deux parties qui contribuent à la construction du graphe : PPHASE1A2 et PPHASE1B (3-1-4);
- les fichiers d'interface (LIST1, ZIN, FILE. GRAPH) et les deux parties de programme qui les génèrent : PPHASE1D et PPHASE1E (3-1-5).

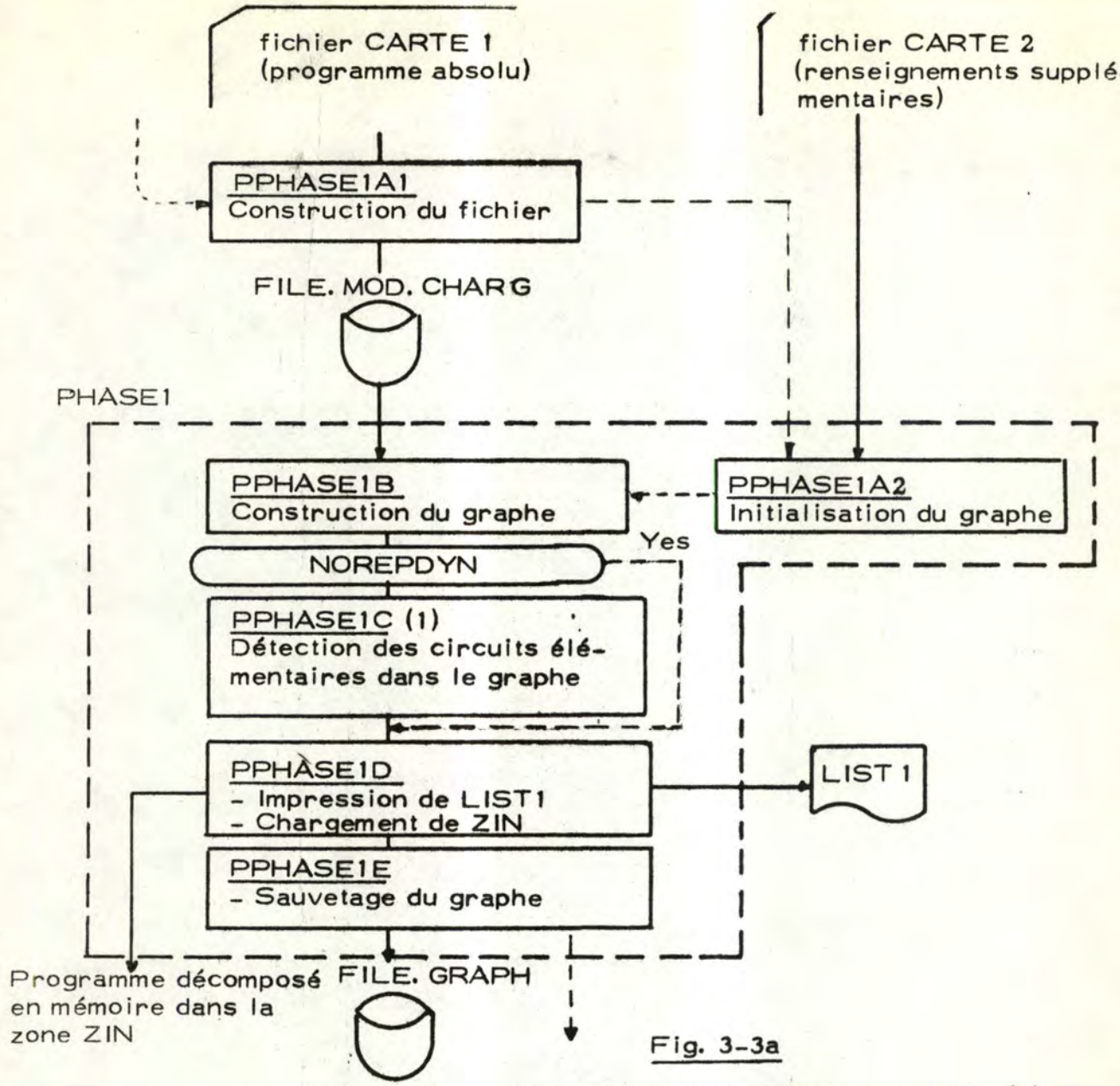



Fig. 3-3a

Structure physique de la PHASE 1 (2)

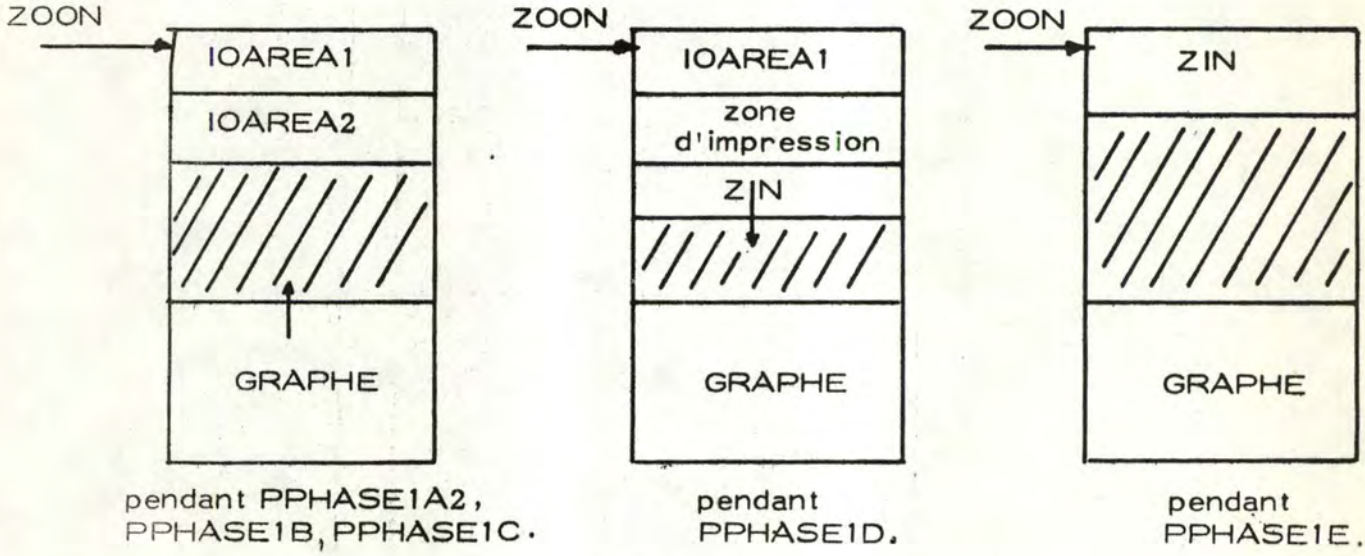



Fig. 3-3b

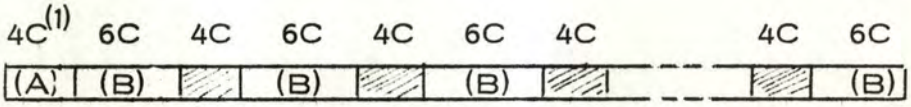
Configuration de la zone commune durant la PHASE 1

---

(1) Partie non implémentée.
(2) ⟶ fichiers employés ou créés
⟶ séquence de l'exécution.

## 3-1-1. DESCRIPTION DES FICHIERS CARTE1, CARTE2.

Ces fichiers contiennent l'un le programme absolu, l'autre les renseignements supplémentaires; le format des enregistrements est donné dans les fig. 3-4 et 3-5; ils utilisent la même zone de lecture qui se trouve dans la zone commune (V. fig. 3-6).



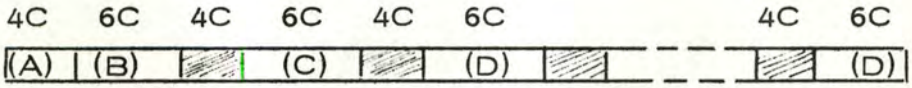

(A) Adresse VARIAN en octal (2) du code décrit par les champs (B) de la carte;

(B) Code en octal (6 caractères octaux = 2 bytes hexadécimaux) ;

fin de fichier détectée par (A) = ***

Fig. 3-4

Description d'un enregistrement du fichier CARTE 1




(A) Adresse VARIAN en octal d'un module externe où l'on branche ;

(B) Nom du module externe;

(C) Nombre de mots VARIAN(3) occupés par les paramètres lors de chaque appel.
(Voir description de l'appel d'un module externe fig. 2-5),

(D) Adresse dans le programme d'un branchement à ce module externe;

fin de fichier détectée par (A) = $$$

Fig. 3-5

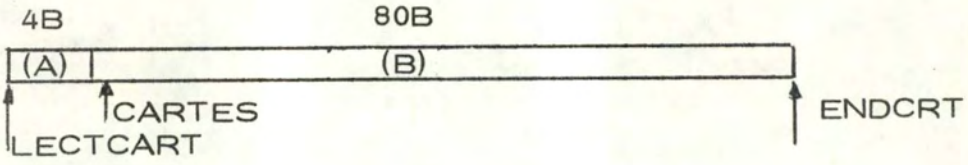
Description d'un enregistrement du fichier CARTE 2

---

(1) C signifie 'caractère', B signifie 'byte', b signifie 'bit'.

(2) l'ordinateur VARIAN travaille sur des mots de 16 bits et tous ses interfaces avec l'utilisateur sont exprimés en octal.

(3) un mot VARIAN contient 16 bits.




(A) Bytes réservés au système ;

(B) Contenu de la carte.

Fig. 3-6

Description de la zone de lecture des fichiers CARTE1 et CARTE 2

## 3-1-2. GENERATION DU FICHIER FILE. MOD. CHARG. PAR PHASE1A1 ET DESCRIPTION DE LA STRUCTURE D'ADRESSAGE QU'IL ENTRAÎNE.

PHASE1A1 traduit (1) le programme absolu se trouvant dans le fichier CARTE1, le 'bufférise' et le stocke par page de 256 bytes (V. fig. 3-7) dans le fichier FILE. MOD. CHARG(2); nous ne nous attarderons pas sur la description de PHASE1A1 qui ne pose aucun problème particulier; nous analyserons plutôt la nouvelle structure d'adressage engendrée par le stockage et sa relation avec l'ancienne.
L'adressage d'un mot dans le programme VARIAN se fait au moyen de la clé de la page et du déplacement dans celle-ci; la correspondance avec le mode d'adressage traditionnel est illustrée à la fig. 3-8.

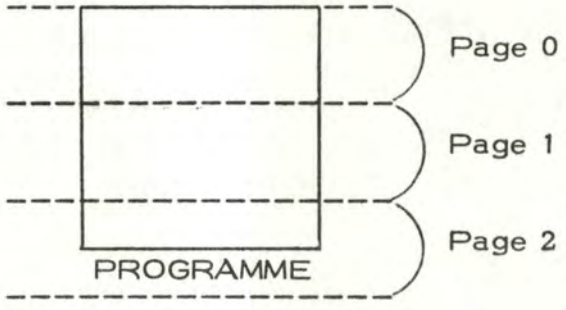



Fig. 3-7
Découpage du programme en pages

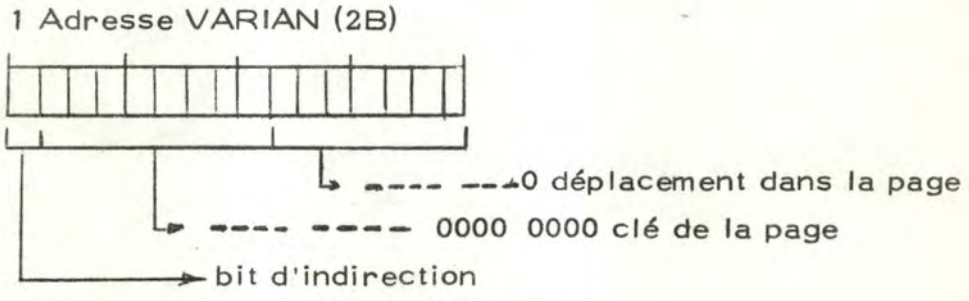



Fig. 3-8
Découpage d'une adresse VARIAN

---

(1) traduction des caractères octaux en bytes hexadécimaux.
(2) fichier indexé séquentiel.

Pour connaître la partie de code se trouvant à une adresse VARIAN, on appliquera l'algorithme suivant :

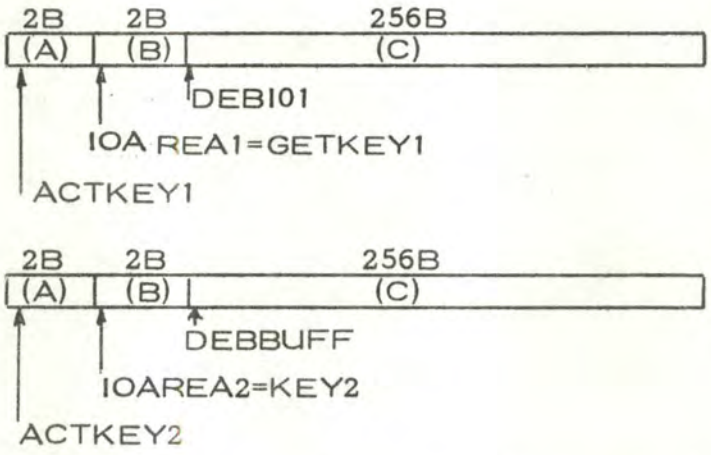
A : à partir de l'adresse, calculer la clé de la page et le déplacement dans cette page (V. fig. 3-8) .

B : si la clé calculée=ACTKEY1 (V. fig. 3-9),
alors aller en C;
sinon si la clé calculée = ACTKEY2,
alors aller en D;
sinon si l'on est en train de résoudre une indirection,
alors mettre la clé calculée dans ACTKEY2 ;
lire la page correspondante; aller en D;
sinon mettre la clé calculée dans ACTKEY1 ;
lire la page correspondante; aller en C.

C : RESULT = DEBI01 + déplacement calculé; aller en E .

D : RESULT = DEBBUFF + déplacement calculé; aller en E .

E : si le bit d'indirection est positionné,
alors aller en A ;
sinon fin du calcul d'adresse ; l'adresse finale se trouve dans RESULT .




(A) clé de la page actuellement présente en mémoire

(B) clé servant lors des accès (n'est pas nécessairement égale à (A))

(C) données proprement dites.

Fig. 3-9

Description des zones de lecture/écriture de FILE. MOD. CHARG

## 3-1-3. REPRESENTATION DU GRAPHE ET DESCRIPTION DES MODULES ASSOCIES A SA GESTION.

Le graphe peut être représenté physiquement de plusieurs manières :

- représentation matricielle qui pour chaque couple de sommet spécifie si la liaison existe (1);
- représentation sommet par sommet qui à chaque sommet associe la liste des sommets suivants.

Dans notre cas, le choix de la représentation sommet par sommet s'impose pour deux raisons :

- un graphe de programme, tel que nous l'avons décrit, possède certaines particularités dont nous devons tenir compte, notamment un faible degré entrant (2) et un faible degré sortant (3) pour chaque sommet; vu les particularités des deux types de représentations physiques décrits ci-dessus et tenant compte du nombre généralement faible de liaisons entre deux sommets du graphe; il semble évident, pour des questions de taille mémoire occupée, de préférer la deuxième représentation;
- de plus, lors de la création du graphe l'évolution de la matrice (V. fig. 3-10) poserait de sérieux problèmes de gestion de mémoire étant donné que la taille maximale de cette matrice n'est pas connue au début de la création.

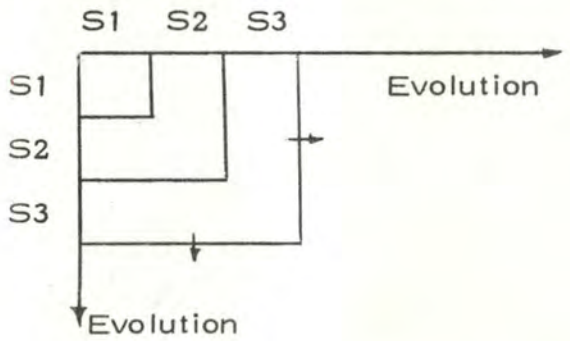



Fig. 3-10
Evolution de la matrice lors de la création du graphe

---

(1) un élément de la matrice spécifie si le sommet correspondant à la ligne a pour suivant le sommet correspondant à la colonne.

(2) le degré entrant d'un sommet est le nombre de flèches aboutissant à ce sommet.

(3) le degré sortant d'un sommet est le nombre de flèches partant de ce sommet; il est au maximum égal à 2.

Le graphe se trouve dans la zone commune de mémoire et à la fin de celle-ci (V. fig. 3-11); lors de sa création, son extension évolue vers les basses adresses; la limite inférieure est en tous temps définie par PTGRAPH.

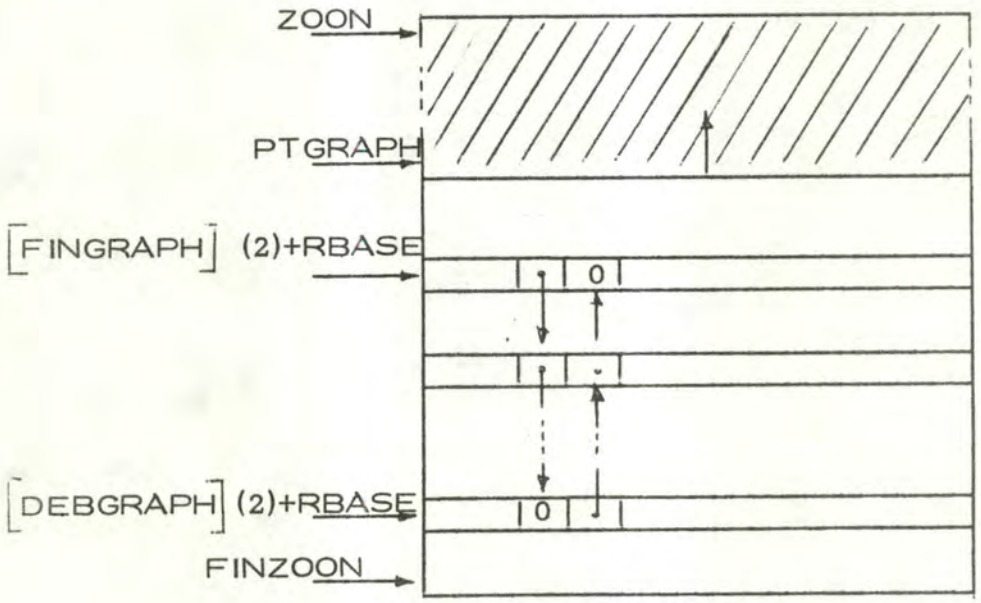



Fig. 3-11

Représentation du graphe dans la zone commune

Dans cette zone sont stockés les éléments représentant les sommets du graphe; leur description est donnée à la fig. 3-12. Les éléments sont chaînés de deux manières :

- chaînages correspondant aux arcs du graphe; ces chaînages se trouvent dans les cases (F) de l'élément;
- chaînages par adresses croissantes et décroissantes (1); nous appellerons cette chaîne, la chaîne principale (2); elle commence en DEBGRAPH et finit en FINGRAPH.

---

(1) les adresses sont bien sûr les adresses dans le programme des instructions correspondant au sommet du graphe; ce chaînage est utile pour :
- choisir l'entrée suivante dans l'algorithme de construction du graphe;
- savoir lors de l'analyse de chaque instruction d'une séquence, si cette instruction n'a pas déjà été vue.

(2) appelée ainsi par opposition à la chaîne secondaire qui relie tous les éléments de type (F) (V. fig. 3-12).

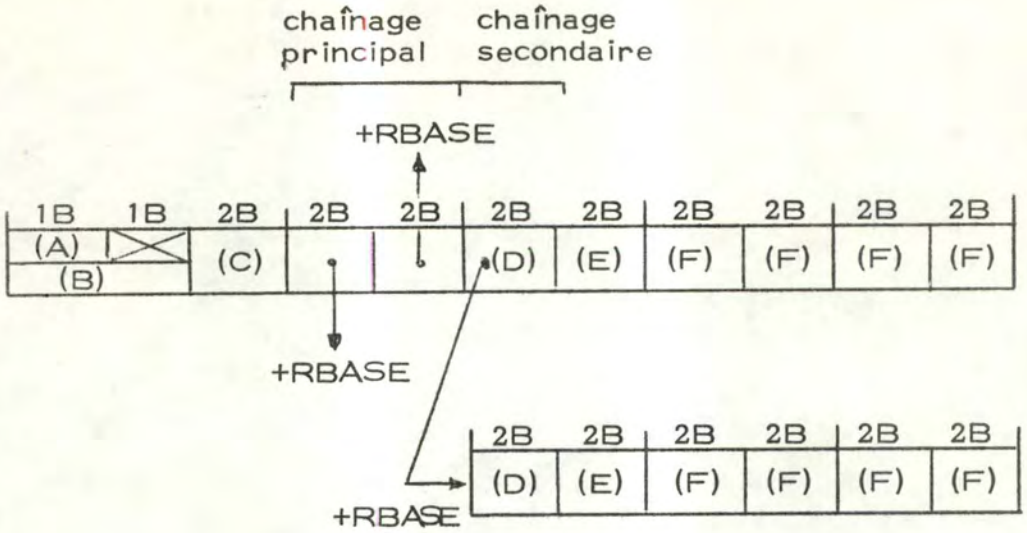
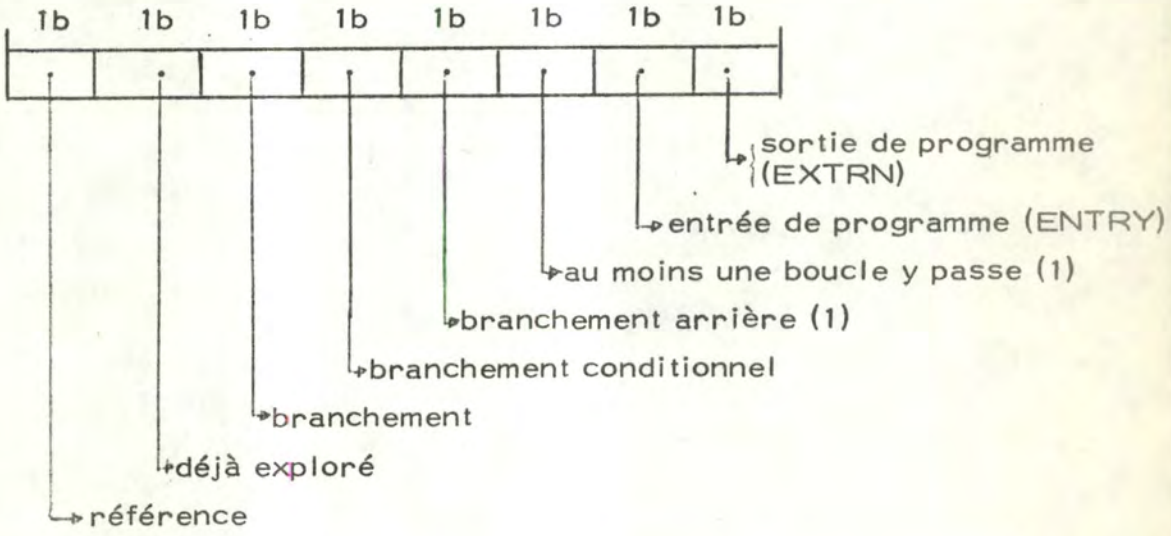



Fig. 3-12

Description d'un élément du graphe

(A) FLAG1 - caractérisent la nature du sommet du graphe correspondant

- donnent certains renseignements lors de la construction du graphe




---

(1) correspond à des extensions futures dans les cas où l'on implémenterait la partie correspondant aux répétitions dynamiques.

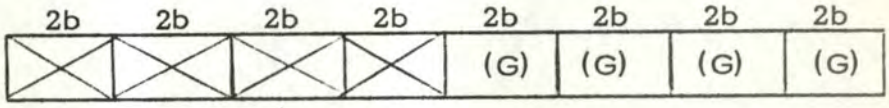
(B) Voir utilisation dans PPHASE1D, sert à faire des conversions d'adresses;

(C) Contient l'adresse VARIAN de l'élément et sert de clé de classement pour la chaîne principale,

(D) Peut avoir deux significations :

- si le nombre contenu est $\leq$ 3, il est égal au nombre d'items de type (F) - 1
- sinon c'est un pointeur de la chaîne secondaire.

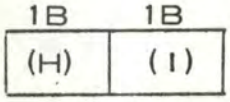
(E) FLAG2




(G) décrit le contenu des éléments de type (F)

00 élément vide
01 adresse où l'on branche (suivant)
10 adresse d'où l'on vient (précédent) (1)
11 numéro de boucle (1)

(F) contient une adresse relative à RBASE dans le cas où le FLAG2 correspondant = 01 ou 10 et contient un descripteur de boucle si le FLAG2 = 11 ceux-ci ont la forme:




(H) est le numéro de boucle (1)

(I) est le niveau d'imbrication de la boucle (1).

La gestion du graphe est assurée par trois modules.

- module RECHCRE :

Etant donnée une adresse VARIAN contenue dans le registre I, ce module recherche dans la chaîne principale l'élément du graphe correpondant à cette adresse;

si cet élément n'existe pas,

alors il le crée à la bonne place dans la chaîne principale et gère les chaînages;

sinon il passe la main au module ELRECH .

Dans les deux cas le retour se fait dans les conditions spécifiées à la fig. 3-13

---

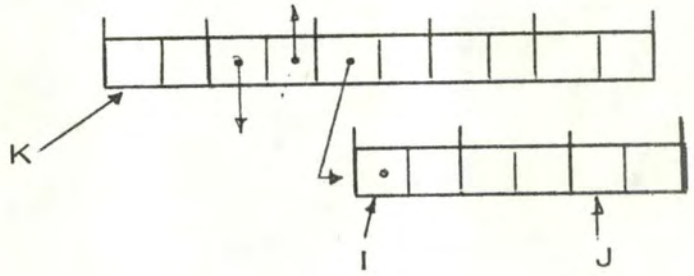
(1) correspond à des extensions futures dans le cas où l'on implémenterait la partie correspondant aux répétitions dynamiques.

- module ELRECH :

Lorsque l'élément de la chaîne principale est connu (K étant l'adresse absolue de cet élément), le module ELRECH détecte le premier élément libre de la chaîne secondaire et gère éventuellement la chaîne secondaire s'il ne reste plus de place; le retour donne les valeurs I et J de la fig. 3-13.

- module STFLAG2 :

Ce module stocke une valeur contenue dans la zone FLAG, dans les FLAG2 correspondant à l'élément désigné par les valeurs de I et J actuelles et normalement positionnées comme à la fig. 3-13.




K = Adresse de l'élément de la chaîne principale

I = Adresse du dernier élément de la chaîne secondaire

J = Adresse du premier item libre de la chaîne

Fig. 3-13

## 3-1-4. PPHASE1A2, PPHASE1B ET LA CONSTRUCTION DU GRAPHE.

La construction du graphe s'opère en deux parties :

- PHASE1A2 prépare le graphe en
  - introduisant l'élément représentant l'entrée principale et en
  - créant les éléments et les chaînages correspondant aux appels de modules externes décrits dans le fichier CARTE2.

  Elle consiste, en fait, en un simple stockage du fichier CARTE2 en employant les modules de gestion du graphe; sa description n'offre donc aucun intérêt particulier.

- PHASE1B construit le graphe à partir du programme en se basant sur l'algorithme logique décrit en 2-2-2; l'implantation physique de celui-ci pose deux problèmes :

  1. pouvoir détecter le moment où le balayage séquentiel du programme doit être interrompu pour faire le choix d'une nouvelle entrée. Deux causes peuvent provoquer cette interruption :
     - la détection d'un branchement inconditionnel;
     - la détection d'une instruction déjà explorée (1).

     Cette deuxième cause nous oblige à connaître à tous moments les éléments du graphe correspondant aux instructions déjà explorées; ce qui peut se faire grâce à un bit dans chaque élément du graphe (2).
     Le positionnement de ce bit peut se faire à deux moments précis :
     - soit au moment de la création d'une référence correspondant à une instruction déjà analysée;
     - soit avant chaque nouveau choix d'une entrée, pour tous les éléments du graphe correspondant aux instructions que l'on vient d'analyser.

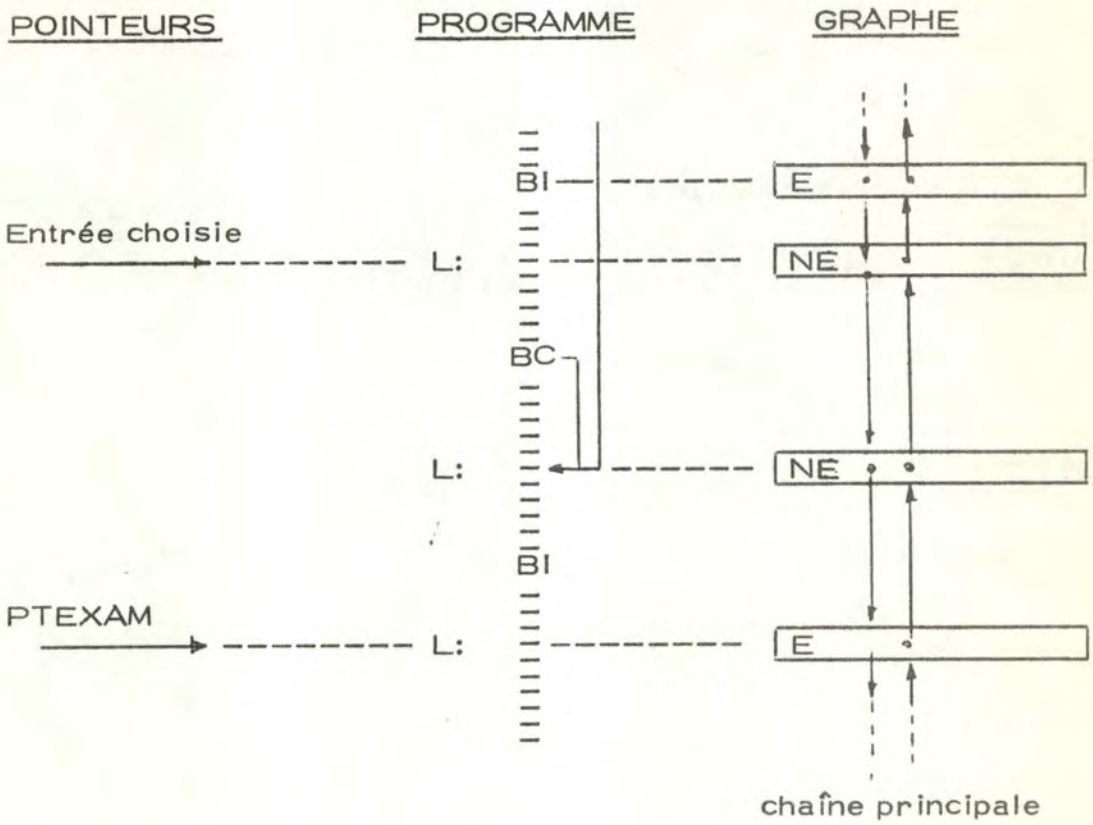
     Pour faciliter la détection des instructions déjà explorées, après chaque choix d'une nouvelle entrée, on recherche (V. fig. 3-14) (3) et l'on stocke dans PTEXAM l'adresse de la première instruction (en séquence) référencée déjà analysée.

---

(1) cette instruction est obligatoirement référencée.

(2) bit 'exploré' qui se trouve dans les FLAG1 (V. fig. 3-12).

(3) cette recherche peut se faire grâce à la chaîne principale.




Légende : L:, BC, BI ont les significations déjà explicitées
E = exploré
NE = non exploré

Fig. 3-14

2. Choisir l'entrée . Le choix tient compte des entrées connues à ce moment et de la position de ces entrées par rapport à la page actuellement en mémoire; il est conçu en vue de minimiser le nombre d'accès disque; la recherche de l'entrée peut être décrite comme suit (V. fig. 3-15) :

A : parcourir les éléments du graphe à partir des plus hautes adresses jusqu'à la fin de la séquence d'instructions que l'on vient d'examiner en vue de découvrir les entrées; sélectionner la dernière entrée trouvée.

B : si on n'a pas trouvé une telle entrée, examiner les entrées possibles à partir du début de la séquence que l'on vient d'examiner jusqu'au début du programme et prendre la première entrée satisfaisante.

C : si aucune entrée n'a été trouvée, le graphe est entièrement construit.

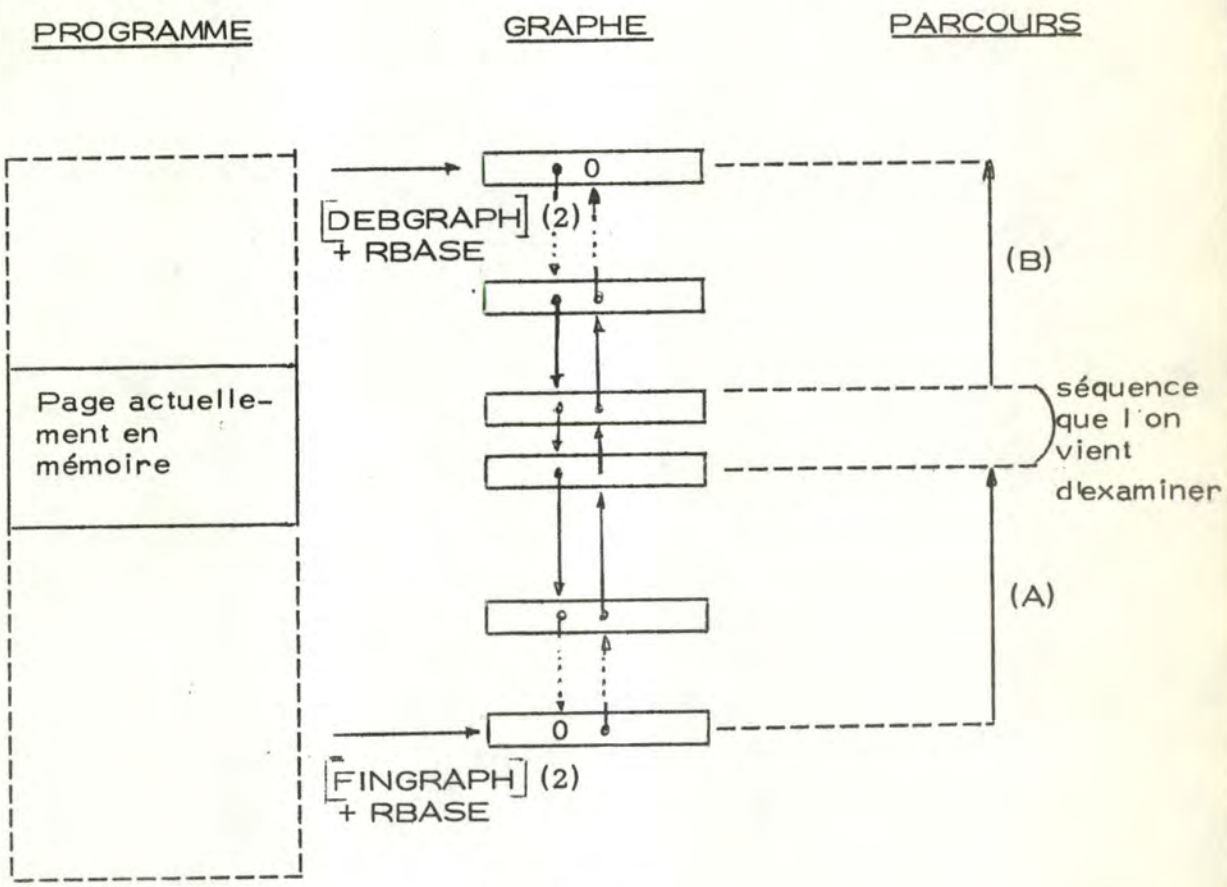



Fig. 3-15

Outre ces deux problèmes viennent se rajouter des problèmes de gestion des appels de pages, reconstitution d'instructions à cheval sur deux pages ...
La structure générale de PPHASE1B est décrite à la fig. 3-16; en vue de la rendre plus compréhensible, nous avons adopté une structure modulaire :

- le module TESTBR analyse l'instruction dont le code opération se trouve dans COINST en vue de découvrir un éventuel branchement, classe ceux-ci en deux types (conditionnels et inconditionnels) et joue le rôle d'aiguillage en fonction du type de branchement détecté.

- le module CALCLG calcule la longueur de l'instruction dont le code opération se trouve dans COINST et charge cette longueur dans LGINST;

- le module BESTCHOI positionne tous les bits 'explorés' des éléments du graphe correspondant à la séquence d'instructions que l'on vient d'analyser, choisit la nouvelle entrée et recherche le 'PTEXAM' correspondant;

- le module REALBRAD calcule l'adresse en cas d'indirection, le nombre d'indirections est à priori, illimité;

- le module BRANCH (1) lance la création et le remplissage des éléments du graphe nécessaires; il utilise pour cela les modules de gestion du graphe (V. paragraphe 3-1-3).

---

(1) appelé lorsqu'un branchement a été détecté.

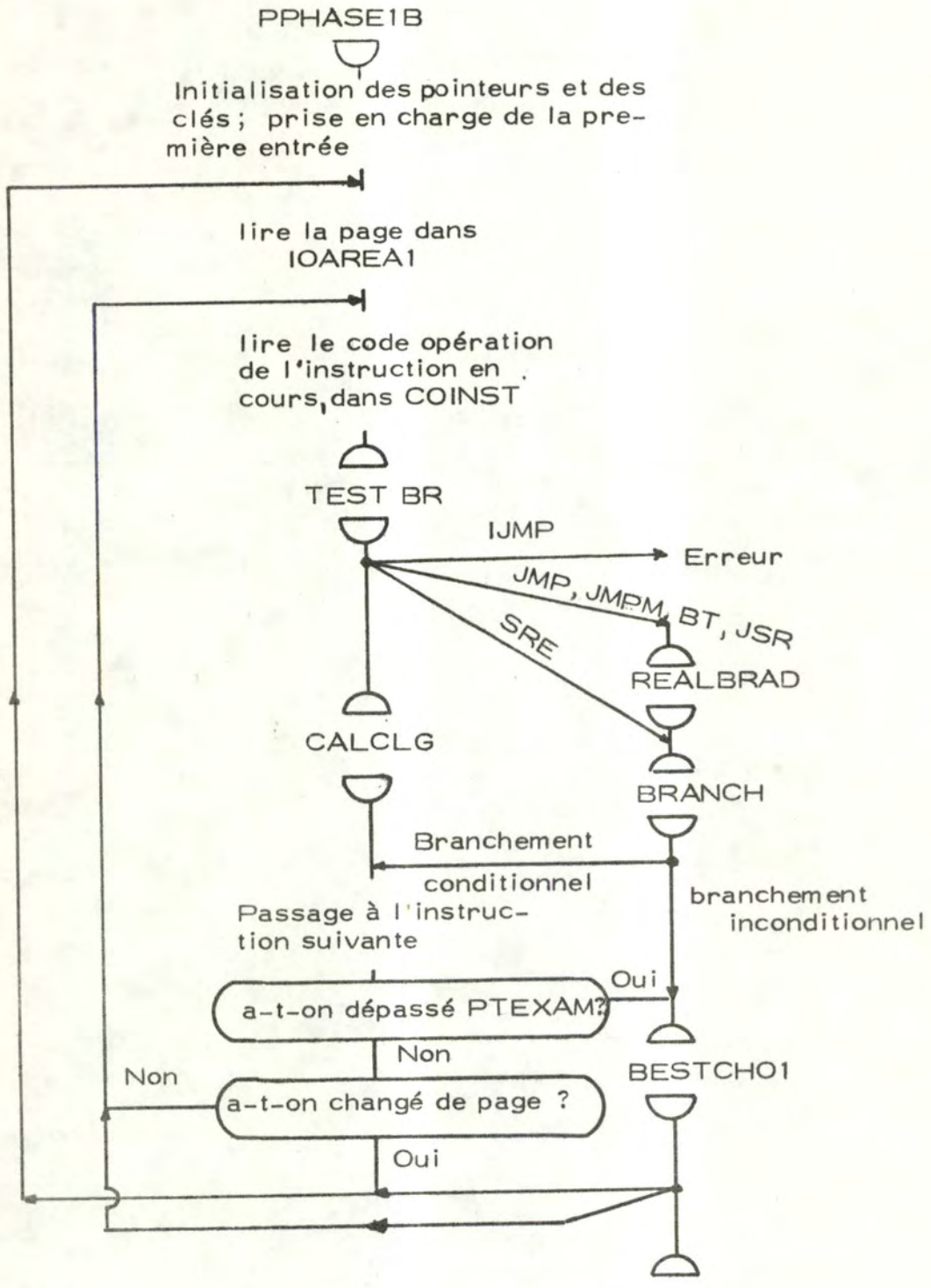
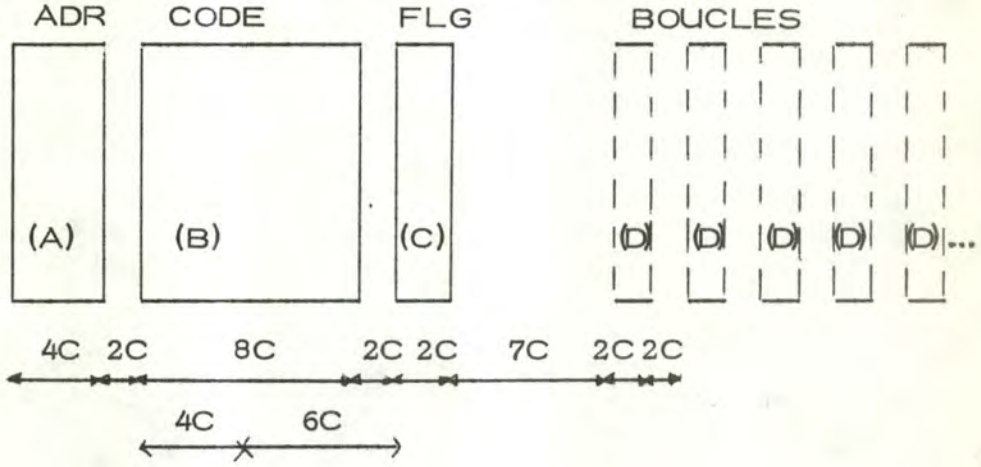

PPHASE1B
Initialisation des pointeurs et des clés ; prise en charge de la première entrée
lire la page dans IOAREA1
lire le code opération de l'instruction en cours, dans COINST
TEST BR
IJMP
Erreur
JMP, JMPM, BT, JSR
SRE
REALBRAD
CALCLG
BRANCH
Branchement conditionnel
branchement inconditionnel
Passage à l'instruction suivante
Oui
a-t-on dépassé PTEXAM?
Non
BESTCHO1
Non
a-t-on changé de page ?
Oui

Fig. 3-16

## 3-1-5. GENERATION DES FICHIERS INTERFACE.

Pour mieux comprendre les opérations effectuées par la PHASE1D et la PHASE1E, il suffit de connaître la structure des trois fichiers interface et leurs relations.

- le fichier LIST1 (v. fig. 3-17) permet à un utilisateur averti de retrouver plus facilement une erreur dans ses données (1) et lui donne en plus un aperçu assez net de la structure de son programme en visualisant les boucles et leurs imbrications.




(A) liste des adresses VARIAN des instructions
(B) liste des instructions VARIAN du programme
(C) flags spécifiant la nature de l'instruction (branchement, label ...)
(D) numéros de boucle.

Fig. 3-17
Fichier LIST 1

- le fichier ZIN (2) contient une liste d'éléments de même longueur (LGCO) qui peuvent être de deux types :

  1 - des codes opérations du programme; nous devons ici préciser la limite exacte entre le code opération et les paramètres d'une instruction du langage machine VARIAN; en effet, si l'on s'accorde généralement pour dire qu'un code opération décrit une opération fondamentale et que les paramètres permettent à cette opération une certaine souplesse, on est loin de pouvoir définir de façon générale et précise la notion de paramètre;

---

(1) une telle erreur peut provenir de causes multiples : erreur de perforation ...
(2) ce fichier se trouve en mémoire au début de la zone commune.

cette notion étant beaucoup trop liée à la structure de chaque langage machine particulier (1); dans le cas du jeu d'instructions machine VARIAN72, nous avons décidé de considérer deux types de paramètres : les adresses mémoire et les paramètres immédiats.(3)

2 - des codes opérations particuliers (v. fig. 3-18) qui remplacent certains codes opérations du programme et ont été introduits pour faciliter le repérage des endroits de ZIN auxquels correspond un sommet du graphe

| | |
|---|---|
| b' 0000 0010 0000 0000' (2) | = branchement inconditionnel |
| b' 0000 001-CONDITIONS' | = branchement conditionnel |
| b' 0000 0000 0000 0000' | = référence devant une instruction normale |
| b' 0000 0100 0000 0000' | = référence devant un branchement inconditionnel |
| b' 0000 010-CONDITIONS' | = référence devant un branchement conditionnel |

Fig. 3-18

Codes opérations particuliers

---

(1) prenons l'exemple d'un numéro de registre; est-ce un paramètre ? Si l'on prend le cas du langage machine IBM360, il est certain que la réponse sera 'oui' ; par contre dans une machine à un accumulateur, on ne considèrera certainement pas l'accumulateur comme un paramètre; dans le cas VARIAN72, le problème est moins évident; les trois registres visibles à travers le langage machine jouent des rôles bien définis dans la plupart des instructions machine; il n'est donc généralement pas possible de les substituer l'un à l'autre sauf dans des instructions telle que le 'load'; nous ne le considèrerons pas comme un paramètre.

(2) b'....' signifie que le contenu est une valeur binaire.

(3) pour plus de détails concernant le format des instructions VARIAN on consultera R24.

- fichier FILE GRAPH : ce fichier contient tous les renseignements contenus dans le graphe et les paramètres des instructions 'normales' (1); on accède au fichier (v. fig. 3-19) grâce à une clé qui n'est autre que l'adresse dans ZIN (relative à RBASE) (v. fig. 3-21) du code opération correspondant; à chaque clé correspond un ou plusieurs enregistrements dont le format est décrit à la fig. 3-20; ces éléments sont chaînés séquentiellement (fichier indexé séquentiel).

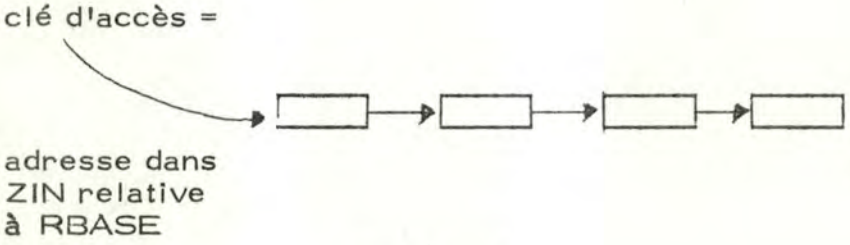
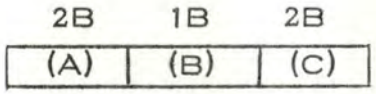



Fig. 3-19




(A) = clé de deux bytes
(B) = nature du contenu de (C); ses valeurs sont :

'A' : le contenu de (C) est l'adresse VARIAN correspondante ;
'P' : le contenu de (C) est l'adresse d'un élément précédent cet élément dans le graphe (adresse dans ZIN relative à RBASE) (2);
'S' : le contenu de (C) est l'adresse d'un élément suivant ;
'B' : le contenu de (C) correspond à une boucle (2);
'V' : le contenu de (C) est un paramètre de l'instruction .

Fig. 3-20

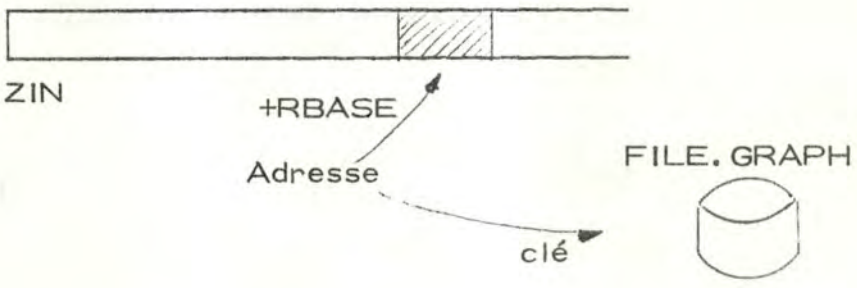



Fig. 3-21

---

(1) c'est-à-dire différentes des branchements.
(2) dans le cadre des extensions prévues.

A partir de ces descriptions, les rôles joués par PHASE1D et PHASE1E paraissent évidents :

- PHASE1D prépare les données à sortir sur imprimante LIST1), remplit la zone ZIN de la liste des codes opérations et met en correspondance les adresses VARIAN et les adresses correspondantes dans ZIN (relatives à RBASE) en stockant dans les deux premiers bytes de chaque élément du graphe l'adresse correspondante de ZIN (v. fig. 3-12);

- PHASE1E traduit toutes les adresses VARIAN en adresses ZIN, prépare les différents enregistrements et les stocke dans FILE. GRAPH.

A la fin de ces deux phases, toutes les adresses sont donc transformées en adresses ZIN et ce sera le seul mode d'adressage dans les phases suivantes.

## 3-2. IMPLEMENTATION PHYSIQUE DE LA PHASE 2 : 'REPERAGE ET CLASSEMENT DES SEQUENCES IMPLEMENTABLES'.

3-2-0. Le passage de la structure logique (V. fig. 2-14) à la structure physique (V. fig. 3-22-a) de la PHASE2, ne pose pas de grosses difficultés, les trois parties logiques se retrouvent respectivement dans (PPHASE2A et PPHASE2B), PPHASE2C et PPHASE2D; nous remarquerons cependant :

- l'éclatement de la partie 'repérage des séquences et classement par types' en deux parties PPHASE2A et PPHASE2B (1); la première initialise les tables en repérant et en classant les séquences de longueur 1; la deuxième étend pas à pas les séquences et les classe par types.

- la séparation entre le traitement proprement dit (repérage et classement) et la création des fichiers de sortie, conçue en vue de clarifier le programme.

Les algorithmes représentant ces différentes phases sont directement déduits des algorithmes logiques en tenant compte de l'organisation des tables en mémoire; celles-ci se trouvent dans la zone commune (ZOON, V. fig. 3-22b) et seront décrites au fur et à mesure des besoins.

Dans les paragraphes suivants, nous étudierons successivement (2) :

- PPHASE2A, PPHASE2B, les tables associées en mémoire et les modules s'occupant de leur gestion (3-2-1),

- PPHASE2C et ses tables (3-2-2);

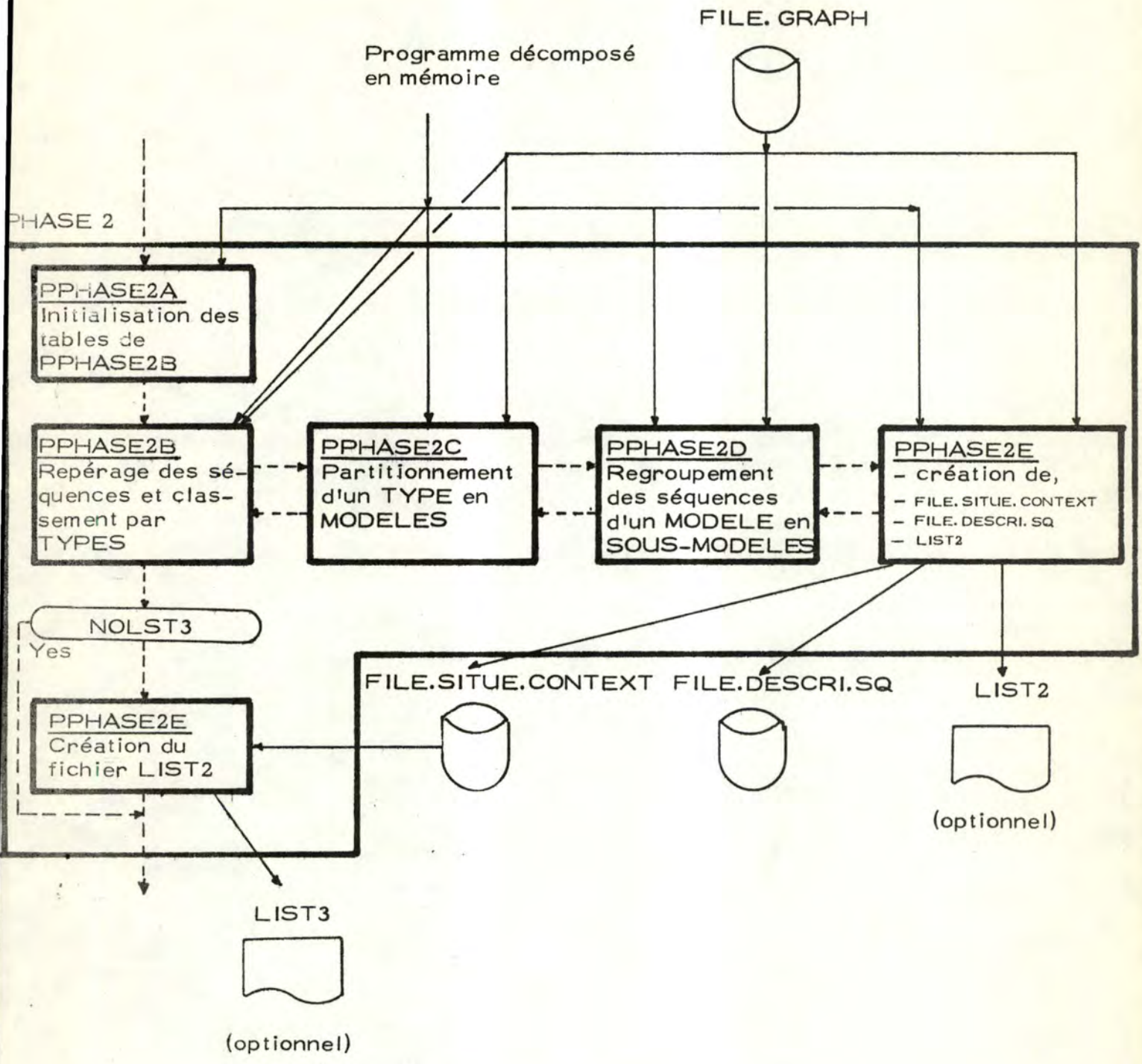
- PPHASE2D et ses tables (3-2-3);

- PPHASE2E, PPHASE2F et les fichiers qu'elles génèrent (3-2-4).

---

(1) l'éclatement était déjà visible au niveau de l'algorithme logique. (V. p. 2-31).

(2) les fichiers d'entrée ZIN et FILE. GRAPH ont déjà été décrits au paragraphe 3- 1 - 5).

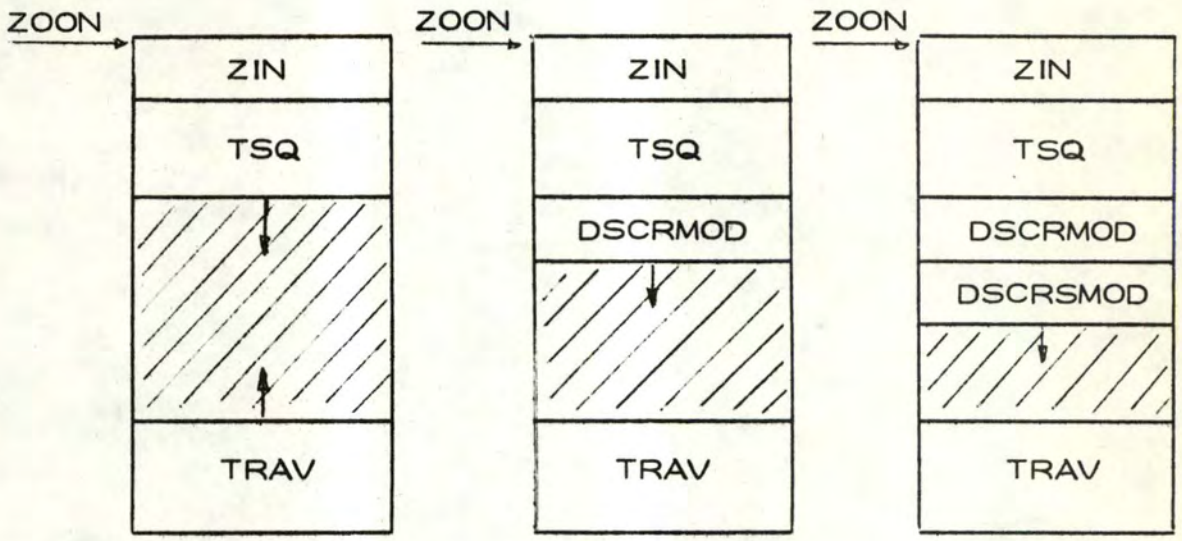



Légende : ——→ fichiers utilisés ou construits

- - - → séquence de l'exécution

Fig. 3-22-a

Structure physique générale de la phase 2




| pendant | pendant | pendant |
|---|---|---|
| PPHASE2A | PPHASE2C | PPHASE2D |
| PPHASE2B | | PPHASE2E (fixée) |

Légende :

| | |
|---|---|
| ZIN | zone entrée de la PHASE2 (programme décomposé). |
| TSQ | table de description des types de séquences. |
| DSCRMOD | table de description des modèles d'un type particulier. |
| DSCRSMOD | table de description des sous-modèles d'un modèle particulier. |
| TRAV | zone contenant des chaînages. |

(ces zones seront décrites dans la suite).

Fig. 3-22-b

## 3-2-1. PPHASE2A, PPHASE2B ET LES TABLES ASSOCIEES EN MEMOIRE.

PPHASE2A et PPHASE2B sont la représentation physique de l'algorithme décrit à la page 2-31.

La configuration mémoire associée (V. fig. 3-22-b) contient en plus de la zone servant de fichier d'entrée (ZIN (1) ), deux tables TSQ et TRAV (V. fig. 3-24) qui sont en fait une simple transposition d'un tableau, semblable à celui utilisé dans l'exemple des fig. 2-15 et 2-16.

Une entrée de la table TSQ et la chaîne qui lui est associée dans la table TRAV représentent une ligne de ce tableau et décrivent donc un type de séquences.
La chaîne contient les adresses (2) des séquences appartenant à ce type (colonne 3 du tableau de la fig. 2-16) et est organisée de manière à réduire le nombre de parcours lors de sa création (3). Un élément de la table TSQ contient en plus du pointeur début de chaîne (V. fig. 3-24) :

- le nombre de séquences qui appartiennent au type (NB) (4) et le niveau de répétition (NIV) dynamique maximum de ces séquences, en vue de faciliter l'emploi du critère de répétition (5);

- le code opération (CO) de la dernière instruction d'une séquence appartenant au type; ce code opération sert de clé de classement des séquences dans l'algorithme (V. p. 2-31) ; il faut en effet se rappeler (6) que le classement ne se fait qu'entre séquences étendues à partir d'un même type, ce qui signifie que ces séquences ont les mêmes codes opérations sauf, peut-être, le dernier.

---

(1) le contenu de ZIN est décrit au paragraphe 3-1-5 et la configuration des pointeurs de cette zone est développée à la fig. 3-23.

(2) adresses dans ZIN relatives à RBASE.

(3) les nouveaux éléments sont rajoutés au début de la chaîne; les seules places libres dans celle-ci, à un moment donné de l'algorithme, se trouvent dans le premier élément (V. fig. 3-24).

(4) c'est-à-dire, le nombre d'adresses dans la chaîne.

(5) ces nombres évitent des parcours de chaîne et des accès au fichier FILE. GRAPH; NIV est prévu en vue d'extensions futures (répétitions dynamiques); il représente le niveau d'imbrication maximum d'une séquence dans une boucle.

(6) le principe de l'algorithme (V. p. 2.31) peut en effet s'énoncer : 'prendre un type de l'étape précédente, étendre de 1 toutes les séquences de ce type et classer les nouvelles séquences par types'.

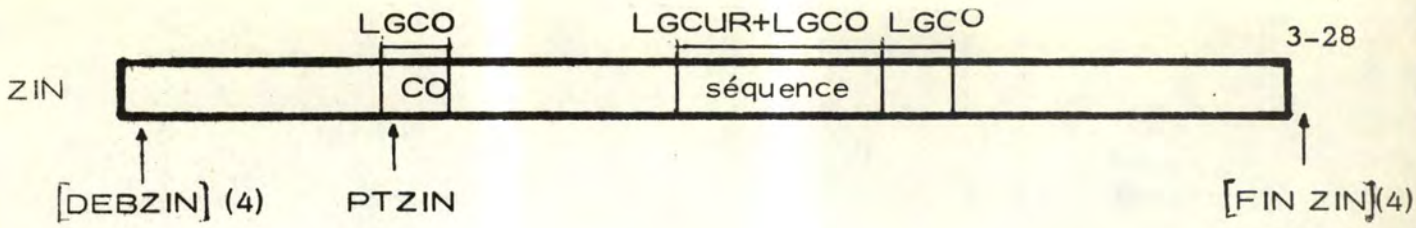



Fig. 3-23

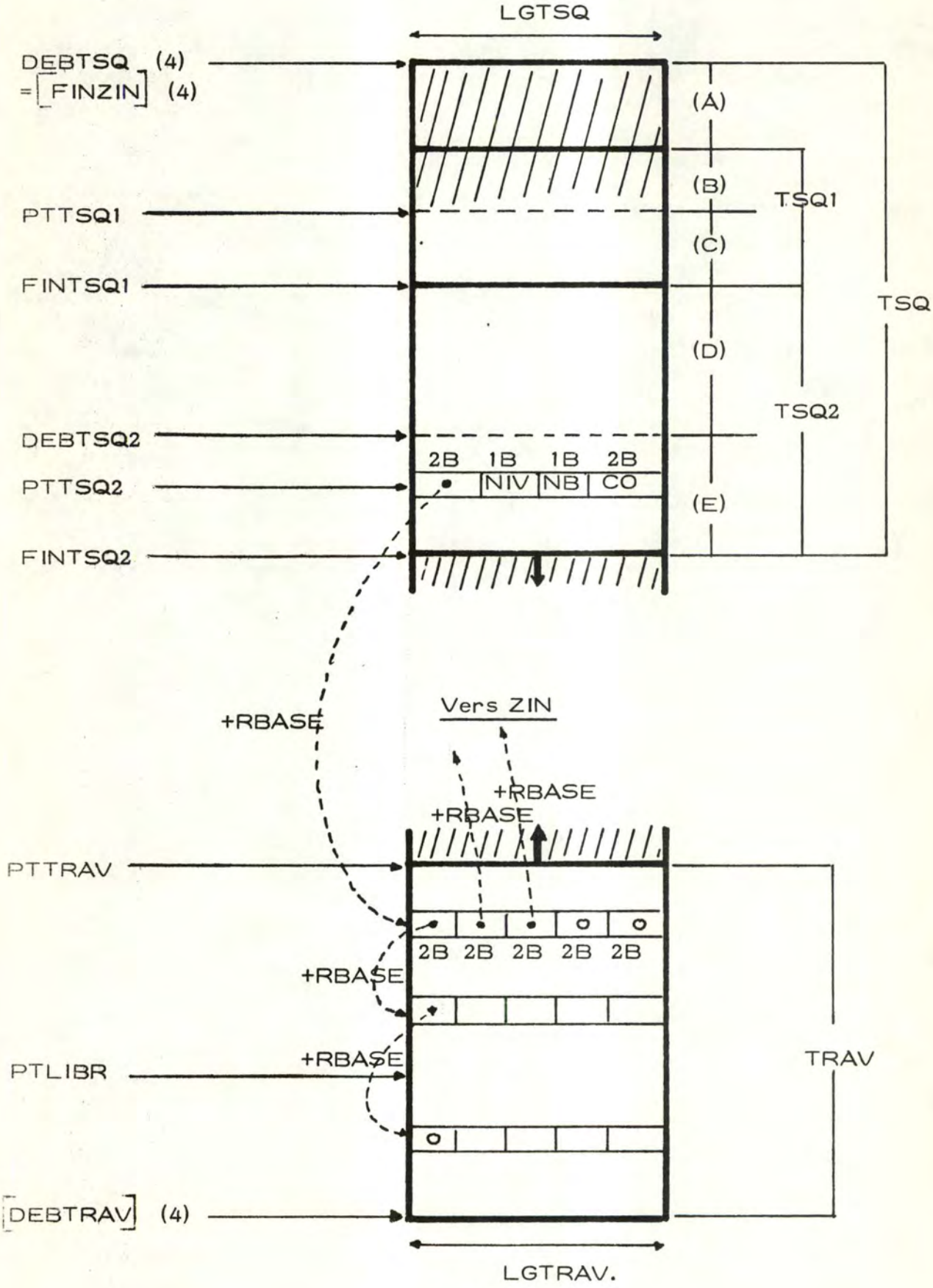



Fig. 3-24

Les tables TSQ et TRAV sont gérées par le module CONSTTAB; étant donné l'adresse dans ZIN d'une séquence (1), ce module recherche dans la partie E (2) de la table TSQ l'entrée ayant la même clé, puis rajoute dans la chaîne correspondante l'adresse de la séquence; il assure éventuellement la création de l'élément de la table TSQ et la gestion des chaînages associés à cet élément.

En réalité, la table TSQ ne reprend pas tout le tableau de la Fig. 2-16, mais seulement les deux parties qu'emploie à un moment donné, l'algorithme, c'est-à-dire :

- la partie du tableau correspondant à l'étape précédente : TSQ1;
- la partie du tableau correspondant à l'étape en cours : TSQ2.

A chaque changement d'étape, le module NVETAPE (3) libère TSQ1 et change la table TSQ2 en TSQ1 par simple modification des pointeurs; en agissant de la sorte, les parties inutiles sont libérées mais non récupérées; les tables TSQ1 et TSQ2 évoluent donc vers les hautes adresses, laissant derrière elles une zone libre (A et B : V. fig. 3-24) (2).

Lorsque l'espace mémoire disponible vient à manquer, (par exemple, lors de la rencontre des tables TSQ et TRAV), le module GARBAG récupère les zones A et B (V. fig. 3-25) par un simple déplacement des zones C, D, E (V. fig. 3-24 et note 2) au début de la table TSQ.

De plus, au fur et à mesure de la libération des éléments dans la table TSQ, les chaînes associées sont récupérées et forment la chaîne libre (pointée par PTLIBR); c'est dans cette chaîne que sont pris par le module CONSTTAB tous les nouveaux éléments nécessaires; le module CHAINVID assure la création de nouveaux éléments dans cette chaîne, ce qui est nécessaire notamment au début de l'algorithme (V. fig. 3-26).

---

(1) l'adresse de la séquence est donnée dans PTZIN, sa longueur = (LGCUR + 1) * LGC0.

(2) à un moment donné de l'algorithme, la table TSQ comprend cinq parties (V. fig. 3-24) :
(A) la zone libérée parce qu'elle correspond à des étapes antérieures;
(B) les éléments libérés parce que les types qu'ils décrivent appartiennent à l'étape précédente et ont déjà subi la procédure d'extension;
(C) les éléments de l'étape précédente qui attendent de pouvoir subir cette procédure;
(D) les éléments déjà créés dans l'étape en cours;
(E) les éléments correspondant aux types en cours de création à partir du type décrit en PTTSQ1;
Ces différentes parties et leurs rôles seront décrits dans la suite de ce paragraphe.

(3) Il incrémente en plus LGCUR DE 1.

Enfin le module REDUCT, appelé chaque fois que toutes les séquences d'un type ont été étendues, détecte les types à abandonner (critère de répétition), supprime les entrées de la table TSQ associées, récupère dans la chaîne libre les chaînes correspondantes, et passe éventuellement (critère de la longueur minimum) la main à PPHASE2C pour créer des modèles...

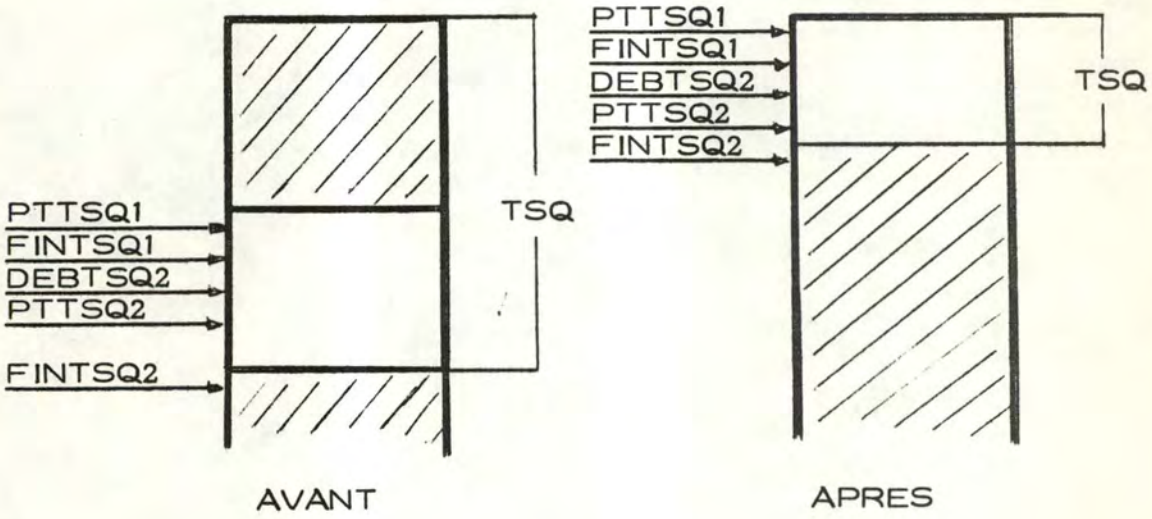



Fig. 3-25

Action du module GARBAG

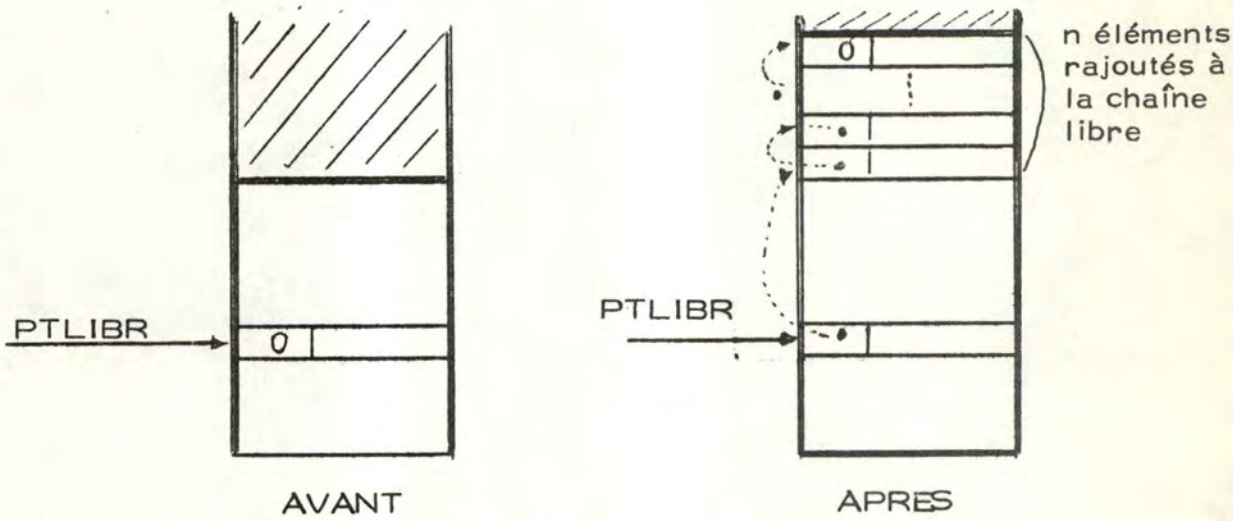



Fig. 3-26

Action du module CHAINVID

PPHASE2A et PPHASE2B sont en fait une suite d'appels à ces modèles et peuvent être définis par les ordinogrammes des fig. 3-27 et 3-28

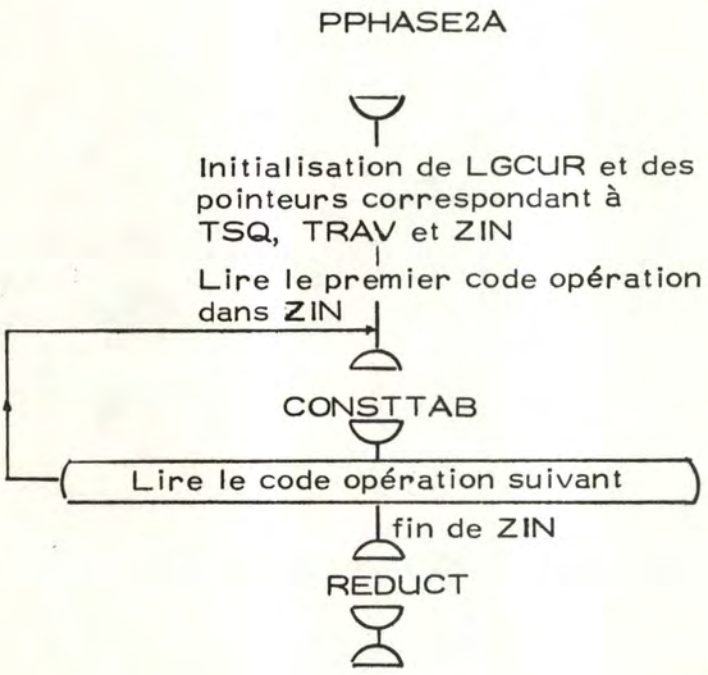



Fig. 3-27

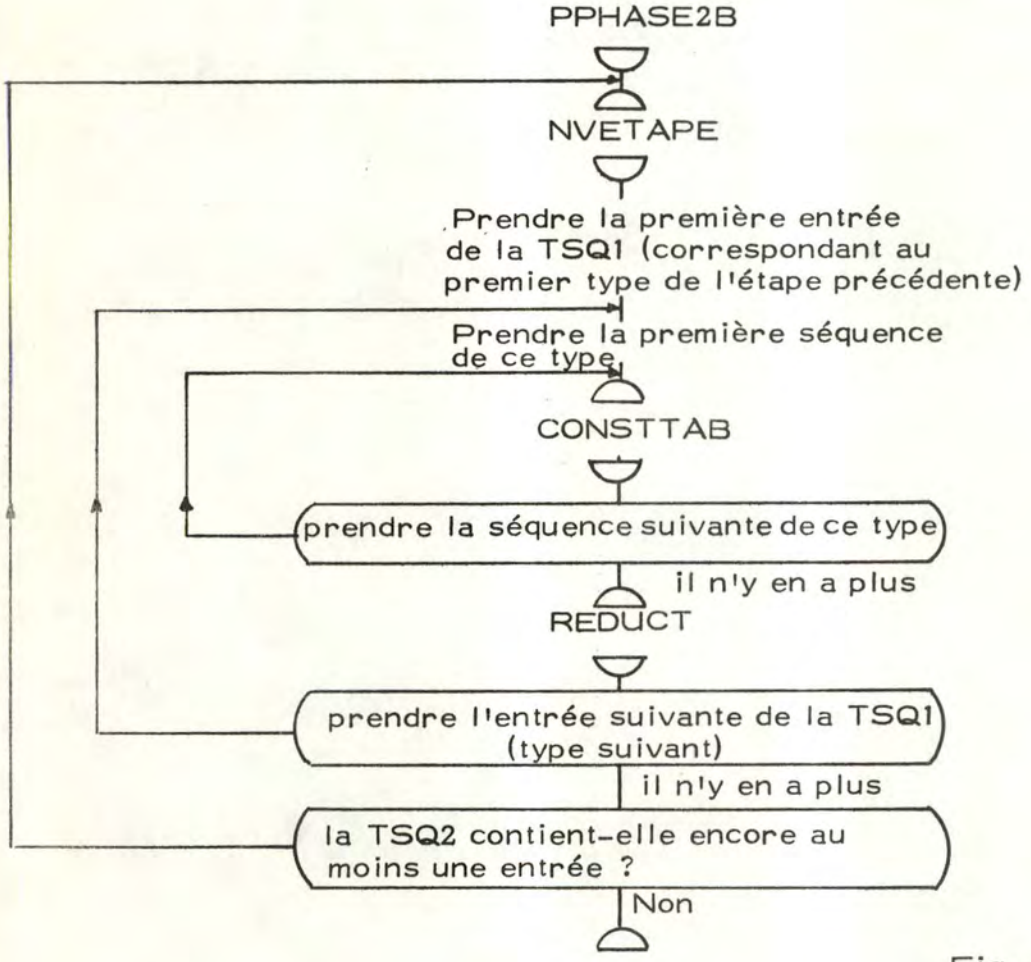



Fig. 3-28

## 3-2-2. PPHASE2C ET SES TABLES.

Cette phase, décrite logiquement à la page 2-32, a pour but de partitionner en modèles, le type décrit par l'élément de la table TSQ pointé par PTTSQ1 (V. fig. 3-29).

Les modèles sont décrits de manière semblable aux types, par un élément dans la table DSCRMOD et une chaîne associée dans TRAV (V. fig. 3-22-b et 3-30).
Cette chaîne contient les adresses des séquences appartenant au modèle et est gérée de la même manière et par les mêmes modules que les chaînes correspondant aux types.
L'entrée dans la table DSCRMOD contient (V. fig. 3-30) :

- LG, c'est-à-dire la longueur de l'entrée (1) et KEY, la clé d'accès au modèle dans le fichier FILE. DESCRI. SQ (2);
- le pointeur début de chaîne;
- les nombres NIV et NB qui ont la même signification que celle employée pour le type;
- la description proprement dite du modèle (champs C).

La création des modèles se passe comme suit :

pour chaque séquence appartenant au type (parcours de la chaîne)(V. fig. 3-29);

- on crée sa structure de branchement (modèle candidat : V. fig. 3-30);
- si aucun modèle homologué n'a les mêmes caractéristiques que le modèle candidat,

  alors on homologue le modèle candidat (3);

  sinon on rajoute l'adresse de la séquence dans la chaîne de ce modèle homologué et on recalcule le NIV et NB;

à la fin pour chaque modèle homologué,

si NIV et NB vérifient le critère de répétition,

alors passer en PPHASE2D; prendre le modèle homologué suivant;

sinon prendre le modèle homologué suivant.

à la fin retourner en PPHASE2B.

---

(1) il faut bien remarquer que DSCR MOD ne contient, à un moment donné que la description des modèles d'un seul type; la longueur des entrées de DSCRMOD est donc constante dans une de ces tables; elle est contenue dans LGMOD.

(2) ces deux zones ont été prévues pour faciliter l'écriture sur disque dans PPHASE2E.

(3) l'homologation consiste en fait en un déplacement des pointeurs (V. fig. 3-30) et à un remplissage de NIV, NB, LG, KEY et de la chaîne.

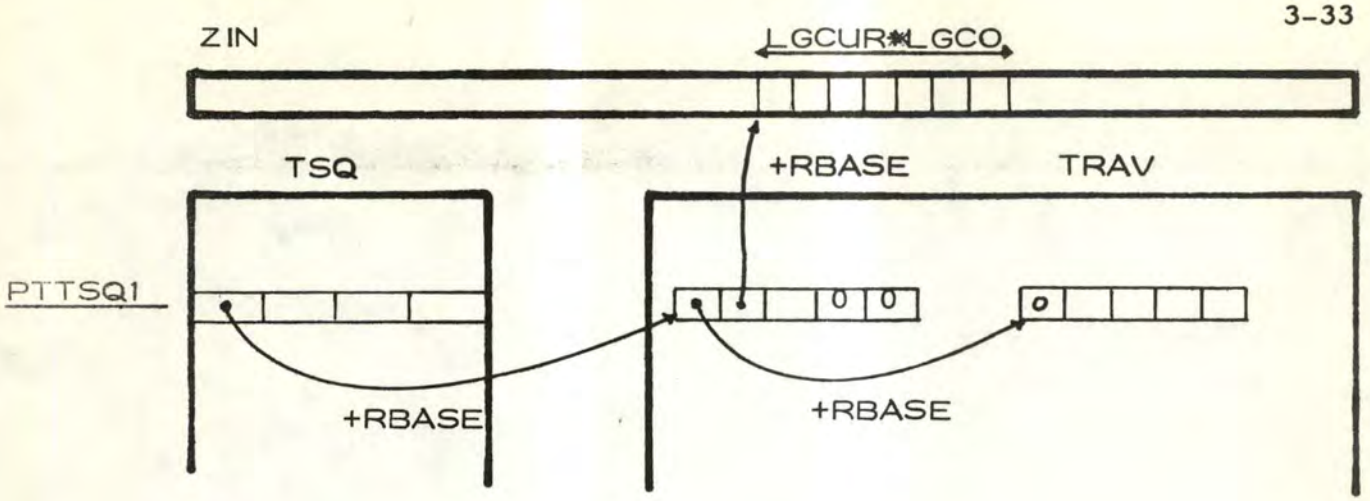
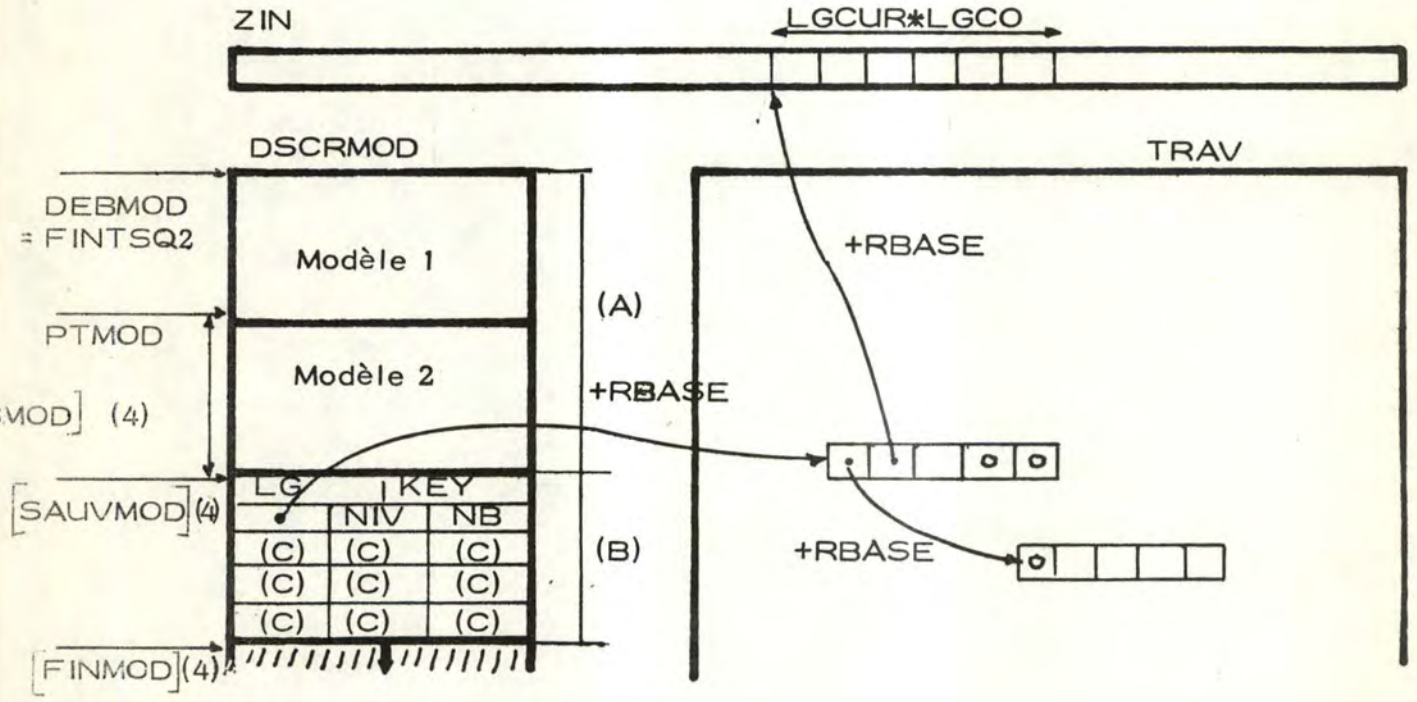



Fig. 3-29




(A) Modèles homologués.

(B) Modèle candidat.

(C) tous les éléments de ce type décrivent la structure interne correspondant au modèle; il y en a autant que d'instructions dans la séquence; la valeur d'un élément est :

- l'adresse de branchement relative au début de la séquence si l'instruction correspondante est un branchement interne;
- X'FF'(1) si l'instruction correspondante est un branchement externe;
- X'00' si l'instruction correspondante n'est pas un branchement.

Fig. 3-30

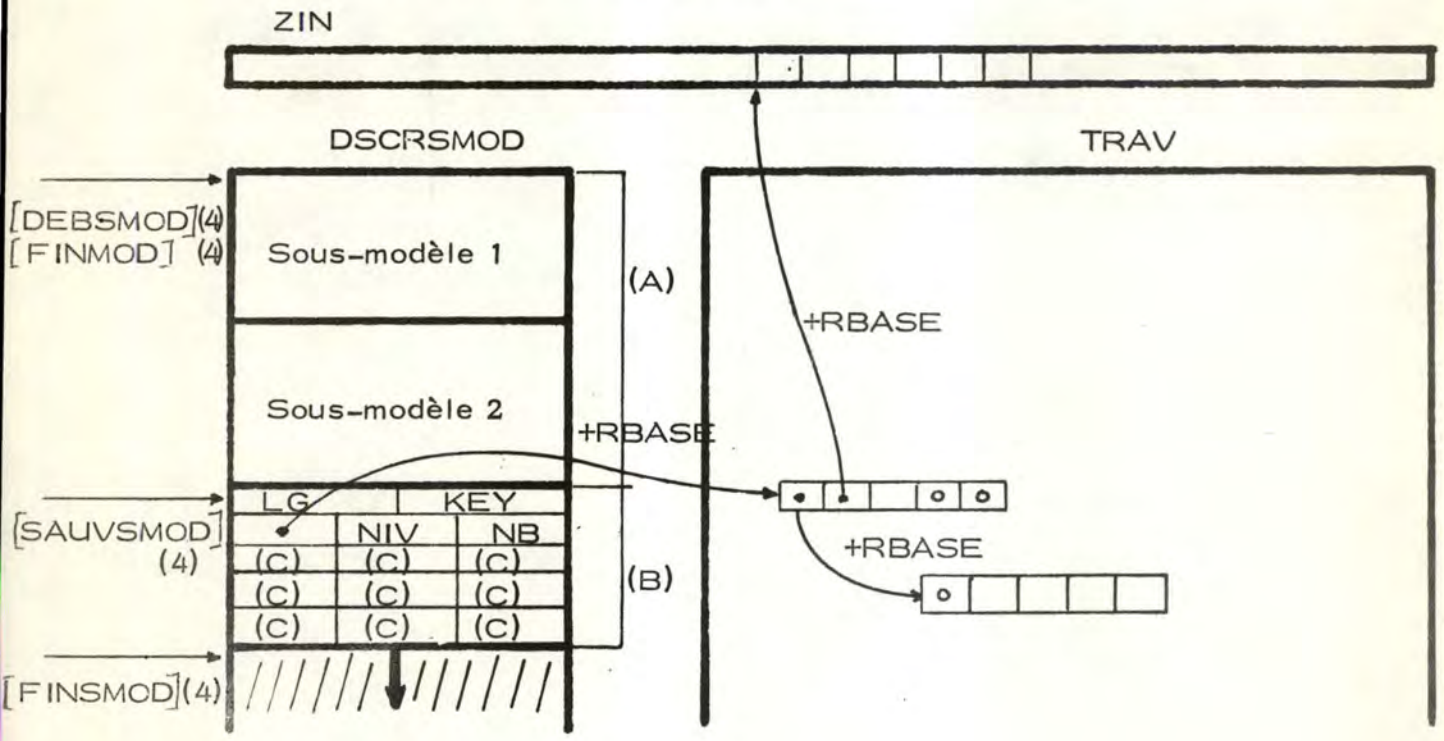
---

(1) X'...' signifie que la valeur représente un Byte et est décrite en hexadécimal.

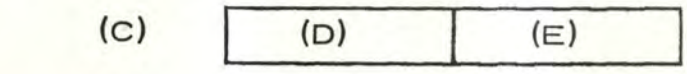
## 3-2-3. PHASE2D ET SES TABLES.

Cette phase décrite logiquement à la p. 2-32, a pour but de regrouper en sous-modèles des séquences du modèle pointé par PTMOD dans la table DSCRMOD (v. fig. 3-30).

L'organisation mémoire des sous-modèles (v. 3-31) est identique à celle des modèles (v. fig. 3-30) et la création proprement dite se fait de nouveau par la constitution d'un sous-modèle candidat et la comparaison avec les sous-modèles déjà homologués; cependant les caractéristiques du sous-modèle candidat sont construites à partir des règles de la page 2-20 appliquées dans un groupe de séquences; tous les groupes possibles de séquences doivent être considérés.




(A) Sous-modèles homologués
(B) Sous-modèle candidat
(C) Ces éléments décrivent la structure des paramètres des séquences du sous-modèle; il y en a autant que d'instructions dans le programme, chaque élément a la structure suivante :



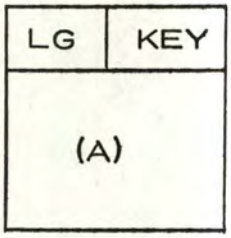

(D) Numéro du paramètre formel (numéroté de 1 à 255).
(E) Valeur du paramètre interne.
(Zéro est la valeur neutre de ces deux champs).

Fig. 3-31

## 3-2-4. PHASE2E, PHASE2F ET LES FICHIERS ASSOCIES.

PHASE2E sauve au fur et à mesure, les zones mémoires qui vont être récupérées; elle construit trois fichiers :

- le fichier indexé séquentiel, FILE.DESCRI.SQ, travaille en longueur variable et contient la description des caractéristiques de tous les types, modèles et sous-modèles.
  Le format des enregistrements est décrit à la fig. 3-32, pour les types; il correspond exactement à celui des tables en mémoire (v. fig. 3-30 et 3-31) pour les modèles et sous-modèles.




légende : - LG et KEY ont la même signification que dans les fig. 3-30 et 3-31
- (A) est la liste des codes opérations caractérisant le type

fig. 3-32

La clé d'accès décrite à la fig. 3-33 est construite au fur et à mesure de l'écriture dans le fichier, grâce à la mise à jour de trois compteurs CMPTTYPE, CMPTMOD, CMPTSMOD.

Elle aura le format suivant :

pour un type : (A, B, C, D) = (i, o, o, o);

pour un modèle : (A, B, C, D) = (i, j, o, o);

pour un sous-modèle : (A, B, C, D) = (i, j, k, o);

pour une combinaison : (A, B, C, D) = (o, o, o, l).

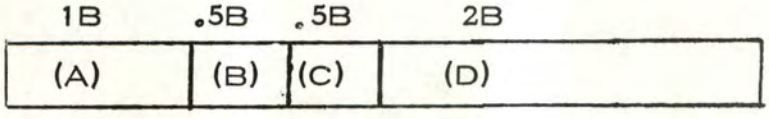



fig. 3-33

Légende : (A) identification du type (=1 à 255)
(B) identification du modèle (=1 à 15)
(C) identification du sous-modèle (=1 à 15)
(D) identification d'une combinaison (voir paragraphe 2-4).

- le fichier indexé séquentiel, FILE.SITUE.CONTEXT, travaille en longueur variable et associe à chaque adresse du programme(1) la liste des numéros de sous-modèles (2) qui contiennent une séquence commençant à cette adresse; pour chaque sous-modèle, il donne la liste des paramètres actuels en correspondance avec les paramètres formels; la fig. 3-34 reprend le format d'un élément de ce fichier.

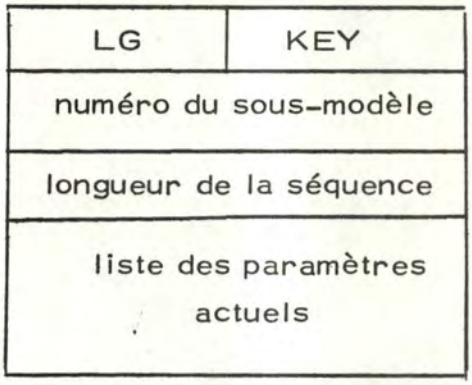



Fig. 3-34

Structure d'un élément

- le fichier imprimante LIST2(3) reprend la description des types modèles et sous-modèles; ce fichier est décrit à la fig. 3-35.

Fig. 3-35

Fichier LIST2

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| IDENTIFICATION : | (A) | | (B) | | (B) | | ... |
| ADR (C) | CODE (D) | (E) | (F) | (G) | (F) | (G) | |
| -- | ---- | ---- | ---- | ---- | ---- | ---- | ... |
| .. | .... | .... | .... | .... | .... | .... | ... |
| -- | ---- | ---- | ---- | ---- | ---- | ---- | ... |
| NB. OCCURRENCES STATISTIQUES: | | (H) -- | | (H) -- | | (H) -- | ... |
| NIVEAU D'IMBRICATION: | | (I) -- | | (I) -- | | (I) -- | ... |

(1) l'adresse dans ZIN relative à RBASE sert de clé (KEY : v. fig. 3-34).
(2) les numéros sont formés comme à la page précédente.
(3) ce fichier peut être refusé par l'utilisateur (NOLST2=YES).

légende : (A) numéro du modèle
(B) numéro des sous-modèles
(C) adresse relative au début de la séquence
(D) code opération
(E) adresse de branchement interne relative au début de la séquence, ou **** dans le cas d'un branchement externe
(F) paramètres formels
(G) paramètres internes
(H) nombre d'occurrences statiques du modèle ou sous-modèle
(I) niveau d'imbrication maximum dans une boucle d'une séquence du modèle ou du sous-modèle.

PHASE2F recopie le fichier FILE.SITUE.CONTEXT sur imprimante (fichier LIST3) (1) (v. fig. 3-36)

| ADR | OCCURRENCES | | |
|---|---|---|---|
| (A) | (B) | | |
| ----- | ---------- | ---------- | ... |
| | ---------- | | |
| ----- | ... | ... | ... |

fig. 3-36 : fichier LIST3.

légende : (A) = adresse dans le programme
(B) = identification d'un sous-modèle.

---

(1) Ce fichier peut être refusé par l'utilisateur (NOLST3=YES).

# CONCLUSION.

Si l'on tient compte de l'évolution actuelle, notre travail se situe plutôt dans le cadre des mini-ordinateurs.

La plupart des ordinateurs actuels utilisent en effet, une hiérarchie de programmation à deux niveaux (1):

- le niveau utilisateur auquel correspond le langage machine et
- le niveau micro-programme,

chaque instruction machine étant interprétée par un micro-programme.

Dans les mini-ordinateurs cependant, une nouvelle voie se dessine; pour compenser les tailles énormes des programmes dues au manque de hardware, on commence à s'orienter vers des hiérarchies plus complexes; c'est la direction que nous avons prise en choisissant trois niveaux de programmation:

- le niveau utilisateur (instructions machine);
- le niveau programme MICALL (instructions MICALL);
- le niveau programme MIDEF (micro-instructions).

L'orientation 'mini-ordinateurs' se marque également dans la manière de résoudre le problème de la répartition des informations dans les différentes mémoires; ce problème joue un rôle déterminant dans les performances d'un système; en principe, trois facteurs doivent être considérés:

- la demande de l'utilisateur (c'est-à-dire les traitements qu'il exécute);
- les possibilités du hardware;
- les capacités et les temps d'accès des différentes mémoires.

---

(1) la configuration est vue sous l'angle 'exécution'; on ne considère donc pas les langages de haut-niveau.

Dans les ordinateurs à deux niveaux de programmation, ce problème se résume en l'adaptation du langage machine aux besoins des traitements.
La plupart des gros ordinateurs utilisent un langage machine choisi arbitrairement et figé. Cette rigidité produit bien sûr, une perte de temps et de place, mais il semble qu'elle ne soit pas critique au niveau des performances du système.
Dans les mini-ordinateurs, le problème est plus délicat; laisser une certaine souplesse au langage machine semble indispensable si l'on veut garder une rentabilité convenable; la solution généralement adoptée consiste alors, à mettre cette souplesse à la disposition de l'utilisateur en lui permettant de créer ses propres micro-programmes et d'adapter ainsi le jeu d'instructions standards à ses propres besoins.
Laisser ce nouvel interface à la disposition d'un programmeur non spécialisé est à la fois dangereux et peu réaliste; nous avons pensé qu'il serait plus sage de laisser à un software spécialisé, le soin de prendre en charge cette souplesse et de la gérer automatiquement en tenant compte des traitements.

C'est dans ce cadre que s'inscrit ce travail, en tenant compte cependant d'une configuration à trois niveaux; deux de ces niveaux restent à adapter; mais leur adaptation simultanée nous a semblé trop complexe pour être réalisée automatiquement; aussi avons-nous choisi la solution intermédiaire suivante:

- nous gérons automatiquement la souplesse du langage machine au niveau du programme;
- nous adaptons manuellement et empiriquement le jeu d'instructions MICALI selon les besoins, cette adaptation pouvant être réalisée au niveau d'un centre pour une période plus ou moins longue.

Le programme réalisé (1) constitue une partie du software assurant l'adaptation automatique.
A brève échéance, outre la poursuite de ce programme, nous comptons réaliser en fonction des besoins une adaptation moyenne du langage machine standard.
A plus longue échéance, lorsque l'adaptation du premier niveau sera devenue automatique, nous commencerons à analyser l'adaptation du deuxième niveau; à ce propos, la seule solution qui nous semble actuellement possible est de partir des modules constituant les instructions machine et de les adapter empiriquement au fur et à mesure des besoins; il est probable qu'une telle adaptation demandera de nombreux tests statistiques.

Les choix que nous avons faits constituent une direction parmi d'autres dans le domaine très vaste des recherches concernant les hiérarchies de mémoires et les répartitions des programmes et des données dans celles-ci.
Il est probable que ces recherches auront un bel avenir dans les mini-ordinateurs, et peut-être qu'un jour les gros ordinateurs s'inspireront eux aussi de ces techniques.

---

(1) ce programme, constitué d'environ 2000 instructions assembleur, est disponible au secrétariat.

# BIBLIOGRAPHIE

## 1. Les grandes voies de recherche en micro-programmation.


R1 AMDAHL
Microprogramming and stored logic
Datamation - V10n2 - (1964) - (p. 24-26)

R2 BROADBENT
Microprogramming and system architecture
Computer journal - V17n1 - (1974) - (p. 2-8)

R3 BROCA, MERWIN
Direct microprogrammed execution of the intermediate text from a high level language compiler
Proceedings of the annual conference of the ACM (1973)

R4 DEMARTEAU
Micro control hardware and high level language interpretor, An attempt of macro supported by firmware
ACM MICRO7 PREPRINTS 7ème Annual workschop on micro-programming - (1974) - (p. 52-58)

R5 DIENEEN, LEBOW
The logical design of CG24
Proc. EJCC - (1958) - (p. 91-94)

R6 DOUCETTE
Performance enhancement by special instruction on system 360/40, 45
Third workshop on microprogramming, Buffalo, New-York (1970)

R7 ECKHOUSE
A high-level microprogramming language M. P. L.
AFIPS conference proceedings - V38 - (1971) - (p. 169-177)

R8 HASSIT, LAGESCHULTE, LYON
Implementation of a high level language machine A. P. L.
CACM - V16n4 - (1973) - (p. 199-212)

R9 LAWSON
Programming language oriented instruction streams
IEEE Trans - C-17 - (1968) - (p. 476)

R10 MALLETT
Approaches to design of high level languages for micro-programming
7ème Annual Workshop on microprogramming (1974)

R11 MELBOURNE, PUGMIRE
A small computer for the direct processing of FORTRAN Statements
Computer journal - V8n1 - (1965) - (p. 24-27)

R12 REIGEL, FABER, FISHER
The interpreteur : a microprogrammable building block system
AFIPS conference proceedings - V40 - (1972) - (p. 705)

R13 ROSIN
Contemporary concepts of microprogramming and emulation
Computing Surveys - V1n4 - (1969) - (p. 197-212)

R14 SALLE, TASSO
Les hiérarchies de mémoires et la technologie
Colloque international sur les mémoires - Paris - (1973)

R15 STEVENS
The structure of the system 360
IBM System Journal - V3 - (1964) - (p. 136-144)

R16 TROMP
Trend on memory characteristics up to 1980
Colloque international sur les mémoires - Paris - (1973)

R17 TUCKER, FLYNN
Dynamic microprogramming processor organization and programming
CACM - V14n4 - (1971) - (p. 240 - 250)

R18 WILKES
The best way to design an automatic calculating machine
Computer inaugural conference : Manchester - (1951) - (p. 16-21)

R19 WILLIAMS
Asymmetric memory hierarchies
CACM - V16n4 - (1973) - (p. 213-222)

R20 WEBER
Microprogrammed implementation of Euler on IBM360/30
CACM - V10n9 - -1967) - (p. 549-558)

R21 B1700 System description
Burroughs Corporation

R22 QM1 Hardware level user's manual
Nanodata corporation Williamsville - New-York - (1972)

## 2. Ordinateur VARIAN 72.

R23 VARIAN 72
Microprogramming guide

R24 VARIAN 72
System Handbook

## 3. Références générales.

R25 BOULAYE
Structure et conception des ordinateurs
Dunod

R26 GRIES
Compiler construction for digital computers
Wiley - New York - (1971)

R27 HUSSON
Microprogramming principles and practice
Prentice Hall - New Jersey - (1970)

R28 KNUTH
Sorting and searching (The art of computer programming : V3)
Addison Wesley - (1973)

R29 MENADIER
Structure et fonctionnement des ordinateurs
Larousse

## 4. Références sur la théorie des graphes.

R30 BAER CAUGHEY
Segmentation and optimization of programs from cyclic structure analysis
AFIPS Conference Proceedings - V40 - (1972) - (p. 23-35)

A31 BERGE
The theory of graphs and its application
Wiley - New York - (1966)

R32 RUSSEL ESTRIN
Measurement based automatic analysis of fortran program
AFIPS - V34 - (1969) - p. 723-732

R33 TIERNAN
An efficient search algorithm to find the elementary circuits of a graph
CACM - V13n12 - (december 70) - (p. 722-727)

5. Références de programmation linéaire.

R34 HU T. C.
Integer programming and networks flows
Addison - Wesley - Massachusetts - 1969

R35 GREENBERG
Integer programming
Academics Press
New York and London - 1971.

---